The Pragmatic Programmers

Broadview®
WWW.BROADVIEW.COM.CN

程序员修炼三部曲　第一部

The Pragmatic Starter Kit-Volume I

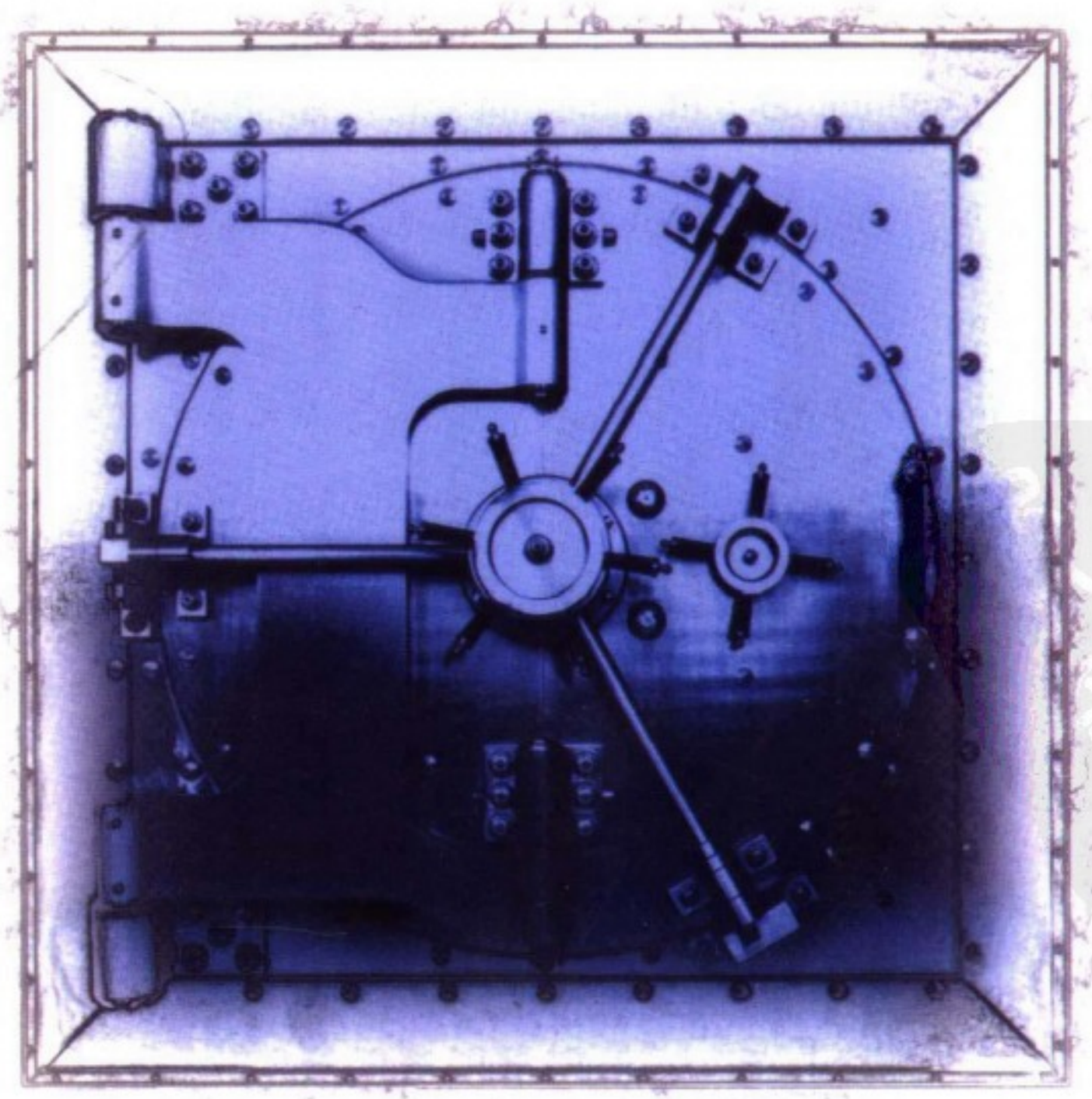
"这是一本关于第一流版本控制系统的第一流书籍。"

—— Martin Fowler

版本控制之道

—— 使用Subversion，第2版

Pragmatic Version Control Using Subversion, 2nd Edition



[美] Mike Mason 著

陶文 译

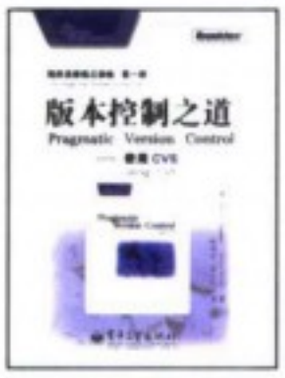
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview
WWW.BROADVIEW.COM.CN
技术凝聚实力 专业创新出版

程序员修炼三部曲

程序员修炼三部曲丛书包含了三个部分，介绍了每个注重实效的程序员和成功团体所必备的一些工具。

您在这里▶
- 第一部曲：版本控制之道
- 第二部曲：单元测试之道
- 第三部曲：项目自动化之道



“版本控制”致力于帮助程序员管理他们的项目资产。它虽然是一门基本的实践性技术，但是许多人并不知道如何使用它，或者未能有效地利用它。与大多数人的想法相反，我们认为版本控制并不是一门难学的技术，更不会是一门难用或者难以实现的技术，而是一门简单的技术，但另一方面，版本控制又是一门不可或缺的技术。如果没有采用版本控制，您的项目资产将会岌岌可危。

本书给出了一系列针对版本控制的方法与技巧，能帮助您更好地挖掘出版本控制系统的各种用处与好处。借助于本书给出的各种方法与技巧，您将能够更好地保护您的项目资产（源代码/文档/测试数据和脚本等），并且知道如何使用这些资产。

本书关注的是广受欢迎的Subversion系统。另外还有一个使用CVS的版本。

阅读本书之后，您将能够做到：

- 共享所有的项目资产（不仅是源代码），并确保安全，绝不让任何一个好的想法丢失。
- 让您可以做更多的实验，并且能够撤销错误的决定——甚至是目录和改名操作都记了版本。
- 安装、管理和备份Subversion项目仓库。
- 让您的项目仓库用svnserve、SSH或者Apache的方式放在网上。
- 有效地组织您的项目仓库，在项目之间共享代码。
- 把现有的CVS项目仓库迁移到Subversion上。
- 使用最新的Subversion 1.3的所有特性，包括加锁和基于路径的安全控制。

Mike Mason是ThoughtWorks的一位咨询师，给全球1000强的公司开发过企业应用。作为一名开发者、敏捷教练以及敏捷/XP的推崇者，他使用版本控制的最佳实践来做开发。Mike对于流行的版本控制系统有着大量的经验，包括Subversion、CVS、Perforce以及Team Foundation。

您可以通过：www.pragmaticprogrammer.com 联系作者。

网上订购：www.dearbook.com.cn 第二书店 第一服务

Pragmatic Bookshelf

策划编辑：方 舟
责任编辑：陈元玉

图书分类：程序设计

ISBN 978-7-121-03768-9

9 787121 037689 >

定价：32.00元

199
:1(2)
2007

程序员修炼三部曲 第一部

The Pragmatic Starter Kit-Volume I

版本控制之道

——使用 Subversion,第 2 版——

Pragmatic Version Control

Using Subversion, 2nd Edition

[美] Mike Mason 著

陶 文 译

電子工業出版社

Publishing House of Electronics Industry

北京，BEIJING

内 容 简 介

《**程序员修炼三部曲**》丛书包含了三个部分，旨在帮助程序员解决在日常工作中遇到的一些具体问题，内容覆盖了对于现代软件开发非常重要的基础性知识。这套丛书展现了注重实效的实际技巧以及工具使用方面的内容。

《版本控制之道——使用 Subversion》是三部曲中的第一部，它讲述如何使用版本控制给整个项目打基础，并从中获取最大的好处和安全性。尽管使用了版本控制会大大提高项目开发工作的效率，但现实中却仍有很多开发小组根本没有使用或不会正确使用版本控制。许多人抱怨版本控制过于复杂，从而对它望而生畏。其实他们只要掌握一些简便的基本用法就可以享有版本控制带来的 90%的好处，而本书正是为了帮助读者从简单处入手，从而比较容易地去掌握版本控制的精髓，达到提高开发工作效率的目的。

0-9776166-5-7 Pragmatic Version Control Using Subversion, second edition by Mike Mason

版权贸易合同登记号 图字：01-2007-0573

图书在版编目（CIP）数据

版本控制之道：使用 Subversion：第 2 版 /（美）梅森（Mason,M.）著；陶文译.
北京：电子工业出版社，2007.3（程序员修炼三部曲　第一部）
书名原文：Pragmatic Version Control Using Subversion, 2nd Edition
ISBN　978-7-121-03768-9
Ⅰ.版...　Ⅱ.①梅...②陶...　Ⅲ.程序设计　Ⅳ.TP311.1
中国版本图书馆 CIP 数据核字（2007）第 004082 号

策划编辑：方　舟
责任编辑：陈元玉
印　　刷：北京市天竺颖华印刷厂
装　　订：三河市金马印装有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编　100036
开　　本：787×980　1/16　印张：16　　字数：240 千字
印　　次：2007 年 3 月第 1 次印刷
定　　价：32.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系电话：（010）68279077；邮购电话：（010）88254888。
质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。
服务热线：（010）88258888。

前言

Preface

当我听说 Pragmatic Starter Kit 的时候非常兴奋，关于那些让项目成功需要用到的东西，终于有这样的书来教大家如何使用了。写 *Pragmatic Version Control* 的 Subversion 版本是我不能错过的机会。Subversion 曾经从地狱里拯救了我（以及我的团队）。我愿意贡献一己之力去推广这个伟大的新版本控制系统。

版本控制能给一个项目带来很多很多的好处。它给你一张安全网，帮助你的团队有效地协作，让你能够组织构建质量保证过程，甚至还可以在有东西出错时让你能够做一些侦查工作。我希望这本新版的 *Pragmatic Version Control* 能够帮助你和你的团队快速上手并成功使用 Subversion。

■ 致谢

我要感谢 Dave 和 Andy 给了我这个机会来写本书，并且要特别感谢 Dave 杰出的编辑工作。有时我都不知道要写一些什么，但是 Dave 的建议和指导给我指明了方向。

本书给许多人仔细审阅过。我要感谢 Brad Appleton，Branko Cibej，Martin Fowler，Steffen Gemkow，Robert Rasmussen，Mike Roberts 以及 David Rupp。他们给了本书许多深刻的评论和建议。坦白地说，我收到的反馈的质量令我非常惊讶。它们中有很好的建议，有很具技术性的评论以及大量的对于“整体蓝图”的思考。

ThoughtWorks 中的每个人对于我的写作都非常支持，一些人还花了不少时间通读了本书的草稿。我要感谢那些给了我建议和指导的朋友，特别感谢

Calgary 办公室今年欢迎我的加入，并且在我忙不过来的时候给了我时间，让我得以完成本书。

最后我要感谢 Martin，Mike 以及 Michelle，是他们让我相信我真的可以写完本书，而且在写作过程中他们一直都在鼓励着我。

2004 年 12 月

■ 第 2 版致谢

自从本书第 1 版问世以来，Subversion 的世界已经发生了很大的变化。它有了新的特性，更好的性能以及更强的稳定性，最重要的是对于很多领先的工具和 IDE，都有了紧密的集成。Subversion 现在可能是 ThoughtWorks 在项目中最常使用的版本控制工具，并且在商业工具市场上也是一个很有实力的竞争者。

自从第 1 版出版以来，许多人给了我支持和反馈，在此我要感谢他们。这么多的人阅读了本书，非常喜欢它，并且 Subversion 给他们带来了成功。知道这些是让人非常满足的事情。请继续给我更多的反馈，它们都很有价值。

以下诸位非常慷慨地贡献了自己的时间来阅读更新后的手稿，并且提供了非常细致的反馈：Steve Berczuk，Nick Coyne，David Rupp 和 Nate Schutta。感谢你们花费的宝贵时间、付出的努力以及提供的极好想法。

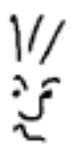
我要感谢 Dave 和 Andy 给了我这个机会来更新本书让其能包容许多 Subversion 的新特性，我特别要感谢 Andy 这次承担起的编辑职责。就如我和我许多朋友和同事说过的，一个好的编辑是写作过程中至关重要的一环，对于能够与 Andy 和 Dave 都有过合作经历，我感到十分幸运。

Mike Mason

May 2006

mike@mikemason.ca

■ 排版标记的规范

黑体	表明这里的名词是正要被定义的名词，或者来自于其他语言的名词，以及给 CVS 用户的提示。
定宽字体	表明这里是方法名称，文件名称，类的名称，或者其他各种常量字符串，以及命令和输出。
	曲线箭头标记表明这些内容是比较高级的，如果你第一次没有看懂的话，可以跳过。
	“开发者 Joe”，他是我们的卡通朋友，在此他会提出一个相关的问题，你或许会发现这个问题非常有用。

读者们是怎么评价《版本控制之道——使用 Subversion》的

虽然读之前期望很高，但是读完之后我甚至比期望的更要来得惊讶。用过很多年 CVS 的我对于是否要使用 Subversion 一直踌躇不决，虽然我知道它会解决 CVS 的许多缺点带来的问题，但是直到读完了你的书之后，我才下定决心迁移到 SVN 上来，所有之前让我留在 CVS 上的借口都烟消云散了。还有，这本书是 Pragmatic 系列之一，读起来也很有趣。多谢了，Mike。

▸ Steffen Gemkow

Managing Director, ObjectFab GmbH

我是 CVS 的老用户，对于 Subversion 一直心存怀疑，不知道它是不是为“黄金时段准备好了”。直到今天，读过了 Mike Mason 这本写得很清楚明白、循序渐进的书对这个新工具的介绍之后，我的疑虑方才消失。实际上，读完本书之后，我对 Subversion 能够给版本控制带来的可能性很是兴奋。

▸ David Rupp

高级软件工程师，Greate-West Life&Annuity

没错，这就是我所期待的那本 Subversion 的书。作为 Perforce 和 CVS 的长期用户兼管理员，以及我自身是一位敏捷工具教练，我想要一本内容紧凑的书来告诉我那些我所需要知道的东西。此书就是这样的一本书。

几个小时内我就把远程 Subversion 服务器给架设起来了，而且我自己的本地服务器也设置好了。Mike 使用了大量的命令行例子来指导读者，说实话，作为一个 Windows 用户，对此我一开始蛮担心的。但是我的忧虑完全是多余的，Mike 的例子太清楚了，使得我从一开始到现在一直都在用命令行！我强烈推荐本书给所有想要使用或者管理 Subversion 的人。

▸ Mike Roberts

CruiseControl.NET项目领导人之一

译者序

译完全书，最大的感触莫过于本书的务实精神。务实体现在本书没有大谈道理，而是处处以实践为根基，用例子说话。务实体现在对很多任务给了多种做法，并分析了各种做法所适合的场合。还有一点我认为是最能体现务实精神的：作者不断地在提醒读者，如果可能，应尽量选择最简单的方案，不要把简单问题复杂化。这种务实精神，也正是敏捷软件开发思想的精髓。

当接到编辑的电话委托我翻译此书时，未曾料到本书的作者竟然是我的同事。后来知道 Mike Mason 是我们 **Thought**Works Canada 的一位前辈，不由得发出一句感叹，难怪！因为在翻译本书的过程中，总是能嗅到熟悉的、审慎的味道。只有每时每刻都想着给客户提供最大价值的人，才能够在字里行间表现出这样审慎的态度。简单的实现才是最完美的实现。**Thought**Worker 都是完美主义者，但他们更是务实的完美主义者。

国内的软件开发行业正处在快速成长期。技术界对于新技术的讨论氛围是非常热烈的。成长中的开发者无可避免地会专情于技术问题，往往思考问题时是技术驱动而不是价值驱动。其实权衡成本和收益不只是老板和项目经理需要考虑的问题。即便你追求的只是成为一流的程序员，你也要学会价值驱动的思维方式。因为对于设计的权衡、技术上的大大小小的决策，往往都能在分析了成本和收益之后找到明确的答案。当你犹豫是否需要一个分支时，问问自己花费多少，收益几何？不创建分支一定会相互干扰么？当你考虑设置 Apache 服务器存放 Subversion 项目仓库时，问问自己是否准备好了维护 WEB 服务器？直接使用 svnserve 是否可行？类似的问题有很多。只要你站在收益的角度来思考问题，很多难题都能迎刃而解。

在本书的翻译过程中，得到了许多人的帮助。特别要感谢 **Thought**Works 的同事们，从他们身上学到的东西让我能够通过文字理解背后的思想。同时还要感谢陈元玉编辑给了我翻译此书的机会，并耐心地完成编辑和审校的工作。最后要感谢我的父母，是他们最初教会了我要如何去做一个务实的人。

陶　文

2006 年 12 月于西安

Pragmatic Starter Kit

Version Control. **Unit Testing**. **Project Automation**. Three great titles, one objective. To get you up to speed with the essentials for successful project development. Keep your source under control, your bugs in check, and your process repeatable with these three concise, readable books from The Pragmatic Bookshelf.

Visit Us Online

Pragmatic Version Control using Subversion
`pragmaticprogrammer.com/titles/svn`
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
`pragmaticprogrammer.com/updates`
Be notified when updates and new books become available.

Join the Community
`pragmaticprogrammer.com/community`
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
`pragmaticprogrammer.com/news`
Check out the latest pragmatic developments in the news.

Save on the PDF

Save more than 60% on the PDF version of this book. Owning the paper version of this book entitles you to purchase the PDF version at a discount. The PDF is great for carrying around on your laptop. It's hyperlinked, has color, and is fully searchable. Buy it now at `pragmaticprogrammer.com/coupon`

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	`www.pragmaticprogrammer.com/catalog`
Customer Service:	`orders@pragmaticprogrammer.com`
Non-English Versions:	`translations@pragmaticprogrammer.com`
Pragmatic Teaching:	`academic@pragmaticprogrammer.com`
Author Proposals:	`proposals@pragmaticprogrammer.com`

The Pragmatic Programmers

每一位程序员的修炼之道！

博文视点外版经典之——

Pragmatic 书架

成套修炼，效果更佳！

《程序员修炼之道——从小工到专家》 Andrew Hunt，David Thomas 著 马维达 译

《版本控制之道——使用CVS》Dave Thomas，Andy Hunt 著 陈伟柱 袁卫东 译

《单元测试之道Java版——使用JUnit》Andrew Hunt，David Thomas 著 陈伟柱 陶 文 译

《单元测试之道 C# 版——使用NUnit》Andrew Hunt，David Thomas 著 陈伟柱 陶 文 译

《项目自动化之道——如何建构、部署、监控Java 应用》Mike Clark 著 张 菲 译

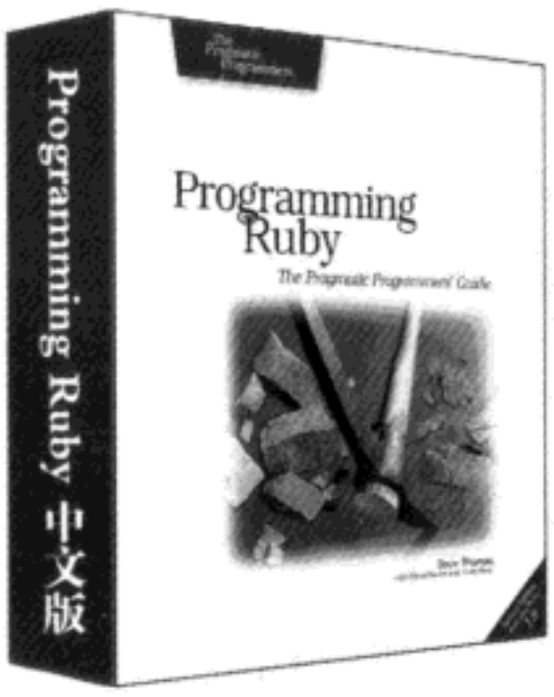
《应用Rails进行敏捷Web开发》

Dave Thomas,

David Heinemeier Hansson 著

林芷薰 译

透 明 审校



《Programming Ruby 中文版》（第2版）

Dave Thomas，Chad Fowler，

Andy Hunt 著

孙 勇 等 译

《Ajax修炼之道——Web 2.0入门》

Justin Gehtland，Ben Galbraith，

Dion Almaer 著

徐 锋 胡 冰 译

《代码大全》是我早在好几年前便已经阅读过的好书。这几年来我不知买过多少书籍，也清理过许多因为书房再也放不下的书籍，但是《代码大全》这本书始终占据着我书架上重要的位置而不曾移开过，因为好书是经得起时光考验的。

——CodeGear公司大中华首席技术官（CTO） 李维

在众多的编程类书籍中，如果只让我挑一本书来阅读，那我一定选择《代码大全》，因为它是最不可或缺的。

——微软亚洲研究院 研究员 潘爱民

能把软件做好的人很多，但是把方法总结出来却很难。《代码大全》做到了，它堪称一本构建软件的百科全书。我在阅读过程中因无数次的共鸣而欣喜。

——《游戏之旅--我的编程感悟》作者 云 风

《代码大使（第2版）》曾伴我走过研习软件开发的岁月，读来收获颇丰，唇齿留香，是技术书中少见的“好吃又有营养”的佳作。

——《JUnit in Action 中文版》译者 鲍志云

《代码大使（第2版）》是最佳的软件构建论著，在中国已经创造了软件开发技术图书出版的奇迹。而我们更希望这个奇迹能够实实在在地推动中国软件开发者能力的提升。

《程序员》 杂志社

Code Complete

代码大全2

中文版
英文版

http://www.cc2e.com.cn

反侵权盗版声明

目录

Contents

第 1 章

简介

Introduction

本书将告诉你如何使用版本控制系统来提高软件开发过程的效率。

版本控制系统，有时被称作**源代码控制系统**，是支撑项目开发的三大支柱之一。在我们看来，所有的项目都必须使用版本控制。

版本控制系统给团队和个人都提供了很多好处：

- 它给团队提供了一个项目级别的撤销按钮：没有什么是最终确定了的，错误可以很容易被回滚。假想一下你正在使用一个世界上最复杂的文字处理工具。它除了不具备“撤销”按钮，但具有所有你可以想到的功能。那就是因为某些原因，开发人员忘记支持删除（DELETE）键了。想象一下，如果是这样的话，你打起字来需要多么仔细，速度会多么慢呀。尤其当你快要完成一篇大文档的时候，出现了一个错误你就要从头来过。对于版本控制来说也是这样的；能够回退一小时、一天或一周，从而可以让你的团队更快地工作，并且有办法回过头来修复发现的错误。

- 它使得多个程序员可以有序地同时为同一个程序写代码。团队不再会因为某人覆盖了其他团队成员所做的编辑而丢失做过的工作。

- 版本控制系统记录着每时每刻的改动。如果你遇到了一些“令人惊讶的代码”时，要找到是谁何时写的就会很方便。幸运的话，你还能知道为什么要这样写。

- 版本控制系统可以让你能够保持主线开发持续进行的同时发布多个版本。有了版本控制系统，就无须在发布之前让团队停止工作以冻结代码了。

- 版本控制系统是一个项目级别的时间机器，可以让你输入一个日期就能看到项目当时的样子。这对于研究来说很有帮助，而且它在顾客有问题要重新产生过往的版本时能发挥重要作用。

本书以项目的视角来关注版本控制，而不是仅仅把版本控制系统中现有的命令一列就了事。它会告诉你要让项目成功须做哪些工作，然后演示版本控制系统是如何帮助你来完成这些工作的。

让我们以一小段故事开始吧……

1.1 现实生活中的版本控制

Fred 冲进办公室急切地想要继续开发那套新的 Orinoco 书籍订购系统（为什么是 Orinoco？那是因为 Fred 的公司总是以河流的名字来命名内部的项目）。在喝完了他的第一杯咖啡之后，Fred 从中心版本控制系统那里更新了项目源代码的最新版本到本地。在日志中列出了更新过的文件，他注意到 Wilma 改动过了基础类 `Orders` 中的代码。Fred 有些担心这些改动会影响他的工作，因为今天 Wilma 去客户那儿安装最新的版本去了，他此时就没法直接问她了。所以 Fred 改为询问版本控制系统，让它显示与 `Orders` 的改动相关的注释。Wilma 的注释让他安心了一些：

```
给Orders类新添加了deliveryPreference字段
```

为了找出具体改动了什么，他回到版本控制系统去观察对源代码实际所做的改动。他看到 Wilma 添加了一些成员变量，但是他们被设置为默认的值，似乎没有什么代码使用了它们。在将来没准这会是个问题，但是就今天来说，对他没什么影响，所以 Fred 继续他的工作。

Fred 写啊写啊，给系统添加了一个新的类以及一些测试类。Fred 在创建这些类的同时把文件的名字添加进了版本控制系统；这些文件本身不会被添加直到他提交了他的改动为止，但是现在就添加它们的名字意味着他晚些时候就不会忘记提交它们了。

几小时过去了，Fred 已经完成了某个新功能的第一部分。它通过了对它的所有测试，而且它没有影响系统其余的任何部分，因此他决定把它签入到版本控制系统中去，让团队的其他人员能够使用它。几年间，Fred 已经明白了频繁地签入签出代码对他来说是最有利的：如果你只有一两个文件，解决偶然的冲突是很容易的。而整个团队同时提交一周的改动时，就不会这么简单了。

■ 为什么你应该绝对不要接电话

就在 Fred 要开始下一轮编码工作之前，他的电话响了。是 Wilma 从客户那儿打过来的。似乎在她正在安装的版本中有一个 bug：打印出来的发票上没有把送货费用计算进销售税。客户很恼火，他们需要现在就解决这个问题。

■ 除非你使用版本控制系统……

Fred 双击了 Wilma 所使用版本的名字，然后让版本控制系统去把这个软件版本的所有文件都签出。他把它放在电脑的一个临时目录下，因为他预计在他修复完 bug 之后就把它删掉。现在在他的电脑系统上就有两套源代码：主干（开发的主线）以及发布给客户的那个版本。因为他要开始修复 bug 了，于是他让版本控制系统给他的源代码打个标签（在修复了这个 bug 之后他还会打另外一个标签）。这些标签就像你留在身后的旗子一样标记着开发工作的关键点。通过在他做出改动的前后都使用命名标签，他的团队中的其他人就能知道都有哪些改动是他们需要关注的。

为了隔离问题，Fred 首先编写了一个测试。似乎没有人曾经在包括送货的情况下检查过销售税的计算，因为他的测试立刻显示出了问题（Fred 做了点记录准备在迭代的回顾会议上提出这个问题：这样的问题是绝对不应该发布出去的）。叹气的同时，Fred 添加了代码把发货的金额添加到计税总额上，编译并检查他的测试是不是都通过了。他把变得全面了的测试以及修正过的代码签入到中心版本控制系统中去。最后，他给发布的那个分支打了一个标签，标记这个 bug 被修正了。他给负责发布紧急版本给客户的 QA 发了封信。使用他的标签，他们可以让构建系统创建一个包含了他的修正代码的发布碟片。Fred 然后给 Wilma 打了电话，告诉她修正代码在 QA 的手里了，应该很快就能到她那里。

Fred 处理完这个小麻烦后，就把发布了的源代码从他的本地机器上删除了。没有必要把东西弄乱，他的改动已经安全地存回中心服务器了。然后他就开始琢磨：是不是在发布的代码中发现的销售税的 bug 在当前的开发版本中也有？最快的检查办法就是把他在发布版本中写的测试添加到开发测试集中。他让版本控制系统把发布版本的那个分支的某些改动合并到开发主线的对应文件。合并过程把所有对发布版本的文件做过的改动在开发版本上再重做了一遍。当他运行这些测试时，他新写的测试失败了：bug 确实还在。他然后把修正代码从发布版本的分支移动到开发版本中来（做这些事情，他都不需要本机上的那个发布版本的分支的任何代码；所有的改动都是从中心版本控制系统那里取来的）。他把所有测试都运行通过了后，接着就将改动签入到版本控制系统之中。这样，下次这个 bug 就不会再出现了。

危机过去了，Fred 回到他今天自己的任务上来。他一下午都很开心地写着代码和测试，直到天色渐晚他决定今天就到这了。在他工作着的同时，团队的其他人也在做着改动，因此他使用版本控制系统把他们的工作取回来并

应用到他本地的源代码上。他最后一次运行所有的测试然后把他做过的改动签入回去，准备好明天接着干。

■ 明天……

不幸的是，第二天又有不同的令人惊讶的事情发生。晚上 Fred 家的中央供暖系统坏掉了。因为 Fred 住在 Minnesota，又是二月份，这可不是好玩的事情。Fred 给上班的地方打了一个电话说今天不去上班了，在家等维修工来修理东西。

然而，这并不意味着他无法工作了。通过使用公共互联网的安全链接，Fred 连接到了他的办公室，接着把最新的开发代码签出到他的笔记本上。因为他前夜在回家之前把代码签入了，所有东西都在并且是最新的。他在家烤着火盖着毯子继续干活。在他即将完成这一天工作之前，他从笔记本上把他的改动签入以让他明天去上班时还能看到它们。生活是美好的（除了修理供暖系统的账单）。

■ Storybook 项目

Fred 和 Wilma 的项目对版本控制系统只是普通的合理运用，没啥了不得的地方，但是它给了他们控制力并且帮助他们交流。甚至当 Wilma 不在时，Fred 也能研究她对代码所做的改动并且给他们的程序的数个版本同时修正 bug。他们的版本控制系统支持离线工作，因此 Fred 获得了某种程度的地点无关性：他可以在遇到供暖问题时待在家里工作。因为他们有版本控制系统（并且知道如何使用它），Fred 和 Wilma 处理了大量项目上的紧急事件，没让客户觉得我们响应不力。

使用版本控制系统给了 Fred 和 Wilma 控制力和灵活性来处理现实生活中的各种突发事件。这些就是本书所要讲的内容。

1.2 路线图

第 1 章介绍了版本控制系统的概念和术语。有许多版本控制系统可以选用。本书中我们会关注在 Subversion 上，它是一个可以从互联网上免费下载的开源软件。Subversion 是最流行的版本控制系统之一的 CVS 的继任者。

第 3 章，“Subversion 入门”是一篇介绍如何使用 Subversion 的教程。本书的其余部分是一些关于在项目中如何使用 Subversion 的实例指导，分为 6 个主要章节。每个章节包含了大量这样的实例指导：

- 以不同的方式连接Subversion。
- 使用常见的Subversion命令。
- 组织Subversion内的文件。
- 使用标签和分支来处理版本和实验性代码。
- 创建项目。
- 处理第三方代码。

本书最后的附录提供一些参考信息和对使用 Subversion 的更深入的讨论：

- 联网，安全以及备份你的项目仓库。
- 迁移到Subversion。
- 使用第三方的Subversion工具。
- 对全文给出的所有实例指导和Subversion的命令的一个归总。
- 使用其他互联网上的资源。

1.3 为什么选择 Subversion

总的来说，虽然本书是关于版本控制的，但是我们选择了 Subversion 作为研究的工具。因为市面上有大量的不同的版本控制工具，可能对于为什么要选择 Subversion 这个问题值得说一说。

Subversion 这个项目是由一个具有丰富 CVS 经验的开发团队发起的(他们中的一些还在这个主题上撰有书籍)。他们认为是该替换掉正在老化的系统的时候了。Subversion 的开发人员对于 CVS 的缺点深有感触，并且确信他们开发的是一个高性能现代化的版本控制系统。他们的目标不是创造版本控制方面的另一种工作方式，CVS 开发模型已经被事实证明是非常成功的了。他们只是用一个新的系统来替换 CVS 去修正所有现有的缺陷。

这可能听起来让人觉得 Subversion 没啥大不了的，但是要知道 CVS 已经比许多其他版本控制工具强得多了，所以 Subversion 的功能完全可以称得上是当今市面上最好的。

■ 对文件，目录和元数据记录版本

目录以及文件是 Subversion 中可以记录版本的对象。这意味着移动或者重命名一个目录是一项“一等公民”操作，目录中的文件自动跟着移动过来了，而且历史信息也正确地被保留了下来。

文件和目录还能以 Subversion 的属性的形式关联相关的元数据。属性既可以是文本也可以是二进制数据，而且也是和文件内容一样被记录版本的，被不断改变，以及被合并到新版本中去等等。属性被大量用来控制 Subversion 处理文件的方式，展开哪些关键字，以及忽略哪些文件，诸如此类。属性的美妙之处在于 Subversion 的客户端可以访问它们，使得第三方工具可以优雅地与你的项目仓库集成。

■ 原子提交和改动集

Subversion 使用了类似数据库事务的方式来处理用户提交代码到项目仓库的过程。整个改动要么成功地被提交要么被中断并回滚。当某人正在提交的时候，其他人是看不到不完整的改动的，你看到的要么是改动之前的状态，要么是改动之后的状态。这样的行为被称为“**原子提交**”。它很有用，因为它能保证每个程序员查看项目仓库时看到的总是相同的东西。如果你正在

提交的途中网络连接中断了，则不会把你的改动留下一半在项目仓库中，所有的改动都会被干净地回滚。

revision

revision number

原子提交过程的其中一步就包括把你的所有改动打包为一个“**修订集**”（有时被称为“**改动集**”），并且给这个改动标记一个修订号。由于对多个文件的所有改动被打包为单一的逻辑单元了，程序员可以更好地组织和跟踪他们所做过的改动。

■ 出色的联网支持

Subversion 有着一个非常有效率的网络协议，并且你的工作文件的原始拷贝是存储在本地的，这使得用户甚至在不连接到服务器的情况下都能查看所做的改动。Subversion 在联网时可以有很多选择，包括使用 Secure Shell（SSH）和 Apache web 服务器让项目仓库可以在公共网络上被访问。

■ 廉价的分支，标记和合并操作

在许多版本控制系统中，创建分支是件很大的事情。比如 CVS，分支或者给代码打标签需要服务器访问并且修改项目仓库中的每一个文件！Subversion 使用了一个高效的数据库模型来分支和合并文件，使得这些操作快速轻松。

■ 真正的跨平台支持

Subversion 在非常多的平台上都有相应的版本，并且最重要的是，服务器在 Windows 上运行得很好。这对于没有 Unix 服务器的团队来说极大地降低了入门门槛，使得它更容易上手。你可以先在一个闲置的 Windows 主机上设置一台服务器（甚至那台机器正在用也没有关系！），等 Subversion 证明了自己的能力之后再迁移到其他机器上。

第 2 章

什么是版本控制

What is Version Control

版本控制系统是一个当你在开发一个程序时储存你写的东西的所有修订版本的地方。本质上来说它们非常简单。不幸的是，这么多年来，人们开始对于版本控制的很多部分使用了不同的术语来描述。这会导致人们迷惑不解。所以让我们从定义会用到术语开始吧。

2.1 项目仓库

你可能已经注意到了我们说“版本控制系统是一个储存……你所写的东西的地方。”但是我们并没有清楚地说明白这些东西会被存放到哪里去。事实上，它们都会被存放到一个叫**项目仓库**的地方去。

repository

几乎在所有的版本控制系统中，项目仓库都是储存各种版本项目文件的主要地方。一些版本控制系统使用数据库作为项目仓库，一些使用的是普通的文件，还有一些同时使用了两者。无论是哪种方式，项目仓库都显然是你版本控制策略中最关键的部分。你需要把它放在一个稳固、安全、可靠的机器上。而且毫无疑问需要定期对它做备份。

在过去，项目仓库和它的所有用户必须共享一台机器（或者至少共享一个文件系统）。这是很大的限制：开发人员在不同的地方，或者使用不同的机器乃至操作系统时就会很困难。因此，大部分当今的版本控制系统都支持联网操作；作为一个程序员，你可以通过网络访问项目仓库，让项目仓库当

不同类型的联网访问

版本控制系统的作者对于什么是联网可能有着不同的定义。对于某些人来说，它意味着通过共享的网络驱动器（例如 Windows 共享或者 NFS 加载驱动器）来访问项目仓库中的文件。对于其他人来说，它意味着有一个客户端/服务器的架构，其中客户端通过网络与服务器上的项目仓库交互。两者都是可行的（虽然前者在底层的文件机制不能可靠地支持加锁的时候会更加难以设计）。然而，你可以发现部署和安全性方面的考虑能很快帮你做出选择。

如果一个版本控制系统须要访问共享的驱动器，而你又要从你的内部网络外面访问它的时候，就必须要确保你的组织允许你以这样的方式访问数据。虚拟私有网络（VPN）能安全地支持这种访问，但不是所有的公司都使用 VPN 的。

Subversion 是使用客户端/服务器的模型来实现远程访问的。

服务器而版本控制工具做客户端，这是相当有价值的事情。开发者在哪里都没有关系；只要他们可以通过网络访问项目仓库，就可以获取项目的所有代码以及这些代码的过去的版本。而且可以很安全地来做这些事情；你甚至可以使用互联网来访问你的项目仓库，不用担心竞争对手会窃取到你宝贵的源代码。

这儿就有一个值得注意的问题了。如果你须要开发但没法连接到你的项目仓库怎么办？简单的答案是，“这要看情况。”一些版本控制系统被设计为仅仅可以在连接到项目仓库时使用；它假定你总是在线的并且在连接到中心项目仓库之前是不能改动源代码的。其他系统的政策就要更宽松一些。本书中我们拿来当例子的 Subversion 系统就是后者中的一员。我们可以在 35 000

英尺的高空用笔记本编辑代码，然后等我们到了宾馆房间之后再把改动同步到服务器。是不是支持离线编辑是选择版本控制系统的决定性因素之一；无论选择什么，都要确保你所选择的产品支持你的工作方式。

一些版本控制系统支持多个项目仓库而不限制说只能有单一的中心仓库。开发人员可以在不同的项目仓库之间交换改动。这被称为“**去中心化的**”版本控制系统。当大量开发人员同时工作时就很需要这样的系统，最著名的例子就是 Linux 内核的开发。去中心化的版本控制系统包括 BitKeeper、Arch 和 SVK 等。这些系统有着非常不同的开发方式，我们在本书中就不对它们作进一步讨论了。

2.2 我们需要存储什么

你项目中的所有东西都被存储在项目仓库中。但是这些东西到底是哪些东西呢?

好的，你显然需要程序的源代码来构建你的系统：Java、C#、Ruby 或者其他你用来写程序的语言。事实上，某些人认为这些源代码对于版本控制来说太重要了，所以他们使用的术语是“**源代码控制系统**”。

源代码当然是重要的，但是许多人往往会犯忘记把其他必需的东西存储到版本控制系统中去的错误。例如，如果你是 Java 程序员，你可能需要 Ant 工具来编译你的源代码。Ant 使用一个一般称作 `build.xml` 的脚本来控制它所要做的事情。这个脚本是构建过程的一部分；没有它，你就不能构建程序，因此它应该被储存在版本控制系统中。

类似地，许多项目使用元数据来驱动它们的配置。这些元数据也应该存储在项目仓库中。还有那些用来创建发布光盘的脚本，QA 用的测试数据，等等。

事实上，要决定放什么不放什么有一个很简单的测试。只要问问你自己“如果我们没有这个东西的最新版本，我们是不是可以构建、测试并交付我们的程序？”如果答案是不能，那么它就应该被放在项目仓库中。

除此之外，对于其他那些会参与到创建发行软件的文件，比如非代码性质的项目产出物也应该用版本控制系统管理起来（简而言之，就是任何你将来需要用到的东西），包括项目的文档（对内对外都包括）。可能还有重要电子邮件的文本，会议的记录，在 web 上找到的信息，总之任何对项目有贡献的东西。

2.3 工作拷贝和操作文件

项目仓库储存了我们项目需要的所有文件，但是光存着还没用，我们要给程序添加功能时该怎么办呢？我们需要把文件放在我们能够修改它们的地方，这个地方被称作我们本地的“**工作拷贝**”。

working copy

工作拷贝是一个我们完成项目当前任务时所需要的一切东西的本地拷贝。内容都是从项目仓库获取而来的。对于中小型项目，工作拷贝可能干脆就是项目的所有代码和其他产出物的一份拷贝。对于更大点的项目，你可能要把东西安排好让开发人员能只在项目代码的一个子集上工作，节约他们花在构建上的时间并且还有助于隔离系统中的各个子系统。工作拷贝也被称为“**工作目录**”或者是“**工作场点**”。

为了第一次建立我们的工作拷贝，需要从项目仓库中拿些东西出来。不同的版本控制系统对于这个过程有不同的叫法，但是最通用的（而且也是 Subversion 所使用的）是叫“**签出**”。当你从项目仓库签出一些东西时，就是把文件复制一份到你的工作拷贝。即便你工作的机器就是存储项目仓库的机器，你仍然要在使用文件之前先把它们签出；项目仓库应该被当成一个黑盒子。签出的过程保证了你得到的是所需文件的最新版本，并且这些文件拷贝出来的目录结构和在仓库中的是一样的。

checking out

\// Joe 问……
产生出来的项目产出物如何处理

如果我们要储存所有用于构建项目的东西，这是不是意味着我们连那些产生出来的文件也要一并存储进去？例如，我们可能会用 JavaDoc 来给我们的源代码产生 API 文档。那样子的文档也需要被存进版本控制系统的项目仓库中去么？

简单的答案是“不需要”。如果一个产生的文件可以从其他文件重新产生出来，那么存储它就是一种重复。为什么这样的重复是不好的？不是因为我们担心浪费磁盘空间。而是我们不希望有东西是不同步的。如果我们同时存储了源代码和文档，那么改动了源代码之后，文档就过期了。如果我们忘记了更新它就把它重新签入了，那样的话，在我们的项目仓库中就有令人误解的文档存在。所以，在这样的情况下，我们希望保持信息只有唯一的来源，那就是源代码。同样的规则也适用于大多数产生出来的项目产出物。

务实地说，某些项目产出物是难以重新产生的。例如，某文件所有的开发者都需要，但是你只有产生它的工具的一个授权，或者每个特定的项目产出物需要花费数小时的时间来创建。在这样的情况下，就有必要把产生出的项目产出物存放到项目仓库之中去。有这个工具的授权的开发人员把文件创建出来，或者放在某个地方的快速的机器创建那耗时的项目产出物。接着签入这些东西，然后所有其他的开发人员都能使用这些产生的文件了。

export

除了签出之外，还能把文件从项目仓库中“**导出**”。它与签出稍微有些不同。当你导出的时候，你得到的不是一个工作拷贝；你得到的只是一个项目仓库文件的快照。这在某些场合下是很有用的，比如打包发行代码时。

committing

当你开发一个项目的时候，会在工作拷贝中修改项目的代码。总有一天你会把修改之后做过的改动又存回到项目中去的。这个过程称为提交。

update

当然，你无时无刻都在做着改动，团队的其他人也是如此。和你一样，他们也会把做过的改动提交到项目仓库。然而，这些改动是不会影响你的工作拷贝的；工作拷贝是不会因为其他人保存了一些改动到项目仓库就突然地被修改了。与此相反，你必须去要求版本控制系统“**更新**”你的工作拷贝。在更新的过程中，你会收到项目仓库最新的文件。而当你的同事执行更新时，他们也会收到你最近刚做的改动。（然而，有些令人迷惑的说法是某些人也把这个更新过程称之为“签出”。因为这个说法很常用，本书中也偶尔会这么说）这些交互过程在图 2.1 中有示例。

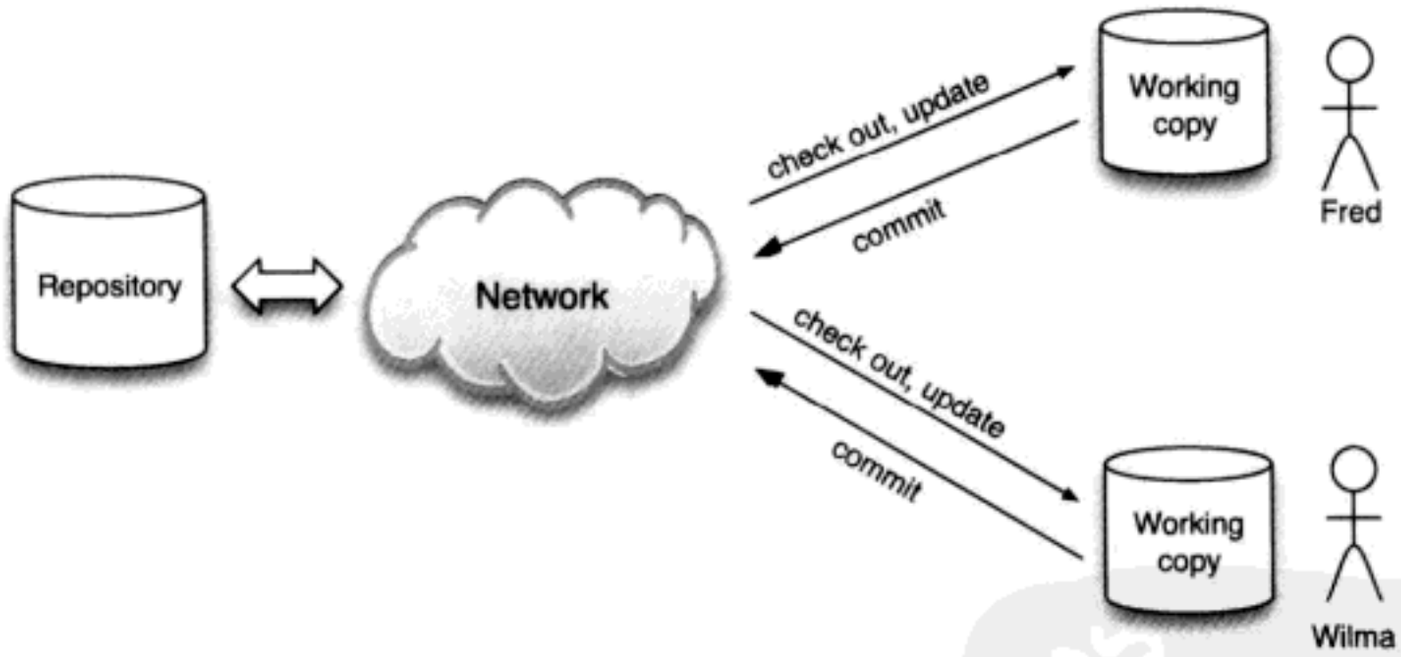


图 2.1 项目仓库和工作拷贝

当然，这儿就有一个潜在的问题了：如果你和你的同事都想同时要改动同一个文件会发生什么事情？问题的答案取决于你使用的是什么版本控制系统，但无论是什么系统，都是有办法处理这样的情况的。我们会在第 23 页的 2.9 节“加锁策略的选择”中深入地讨论这个问题。

2.4 项目，目录以及文件

到目前为止，我们谈论过了储存“东西”方面的事情，但是我们还没有谈论到这些东西是如何组织起来的。

在最低的层次，大部分版本控制系统处理的都是单独的一个个文件[1]。你项目中的每个文件按照名字存储在项目仓库之中；如果你添加了一个叫 `Panel.java` 的文件到项目仓库之中，然后团队中的其他人就可以用 `Panel.java` 的名字签出到他们自己的工作拷贝中。

然而，这是相当底层的做法。一个典型的项目可能会有成百上千个文件，而且一个典型的公司可能会有数十个项目。幸运的是，几乎所有的版本控制系统都有组织项目仓库的办法。在最顶层，它们一般把你的东西分成不同的项目。每个项目中，你还可以划分出模块（有时也叫子模块）。例如，假设你在做一个叫做 Orinoco 的项目，它是一个大型的基于 web 的图书订购程序。构建程序的所有需要的文件都存储在项目仓库中的 Orinoco 项目名下。如果你需要的话，你可以把它们都签出到你的本地磁盘上。

Orinoco 项目自身可能会被分解为数个基本上互相独立的模块。比如，可能有一个团队去做信用卡处理那块，另外一个团队去做订单处理。幸运的话，做信用卡子项目的人不需要项目所有的源代码就可以做好他们自己的工作；他们的代码应该被很好地隔离开。因此当他们签出的时候，他们希望看到的只是项目中正在做的那部分。

[1] 一些类 IDE 的开发环境在方法级别进行版本控制，然而并不常见。

externals

Subversion 以目录的形式组织项目仓库。一个项目可能对应的是一个顶层的目录，模块和子模块则是你项目中的目录。对于简单的项目来说，这可能够用了。但是为了满足更大规模共享代码的需要，Subversion 也支持“外部引用”。一个外部引用就是把另外一个 Subversion 项目仓库的位置包含到你项目的任何目录中。

给 CVS 用户的提示：Subversion 的基于目录的组织方式简单来讲就是 CVS 的模块，而外部引用就是“模块别名”。通过目录来组织被证明为是功能强大的，同时用户理解起来也不困难。

Subversion 的“万事皆为目录”的做法在第 111 页的第 8 章“组织你的项目仓库”中有更深入的讨论。

2.5 版本从何而来

本书全是讲版本控制系统的，但是到目前为止，我们谈论的都是关于如何从项目仓库中存取文件的事情。那版本从何而来？

其实在幕后，版本控制系统的项目仓库是一个相当聪明的家伙。它不仅仅存储所有文件的当前状态，而且会存储曾经签入的“**每一个版本**”。如果你签出一个文件，编辑它，然后再签入回去，项目仓库既会保存原来的版本也会保存你改动过的版本。实际上，大部分版本控制系统储存的只是文件不同版本之间的差异的部分，而不是把文件的每个修订版本都完整地存下来。Subversion 储存文件的最新版本，同时也会聪明地选择恰当的历史版本存储其全文，从而无论是取文件的哪个版本都能很快地获得。这样的设计使得在最小化磁盘空间的同时更新和签出速度也够快。

关于对版本的编号，有两种常见的方式：**具体文件编号和项目仓库整体编号**。在具体到文件的编号方式下，文件的第一个版本是 1.1。签入了一个改动后，文件版本被编号为 1.2 了，依此类推。如果你有一个 1.2 版本的

Node.cs 和一个 1.6 版本的 Graph.cs，提交一个对 Node.cs 的改动之后，其版本号变为 1.3，而 Graph.cs 的版本号不会变，仍然是 1.6。

在项目仓库整体编号的方式下，整个项目仓库从版本号 0 开始，签入一个改变后，项目仓库的版本号被增加为 1，然后是 2，依此类推。若是使用这种编号方式，更恰当的说法是“版本 7 中的 Panel.java”而不是说“第 7 版本的 Panel.java”。Subversion 使用了第二种编号方式，事后被证明对于引用提交过改动的文件是极端有用的。第 125 页的 9.5 节“修正简单的 bug”讲述了如何使用版本号在不同分支之间合并修正 bug 的代码。

给 CVS 用户的提示：CVS 使用的是具体文件编号方案，因此人们经常去看文件的版本号以了解对这个文件做过了多少次改动。因为 Subversion 使用的是全项目仓库的编号方式，所以就不能这样做了，你必须使用 Subversion 的日志命令来查看历史记录以了解改动的情况。

Subversion 的项目仓库版本号有点像马克笔，在每次提交的时候给项目仓库中的所有文件画一条线。图 2.2 中有三个文件：Trains.java，Graph.java 以及 Node.java。首先我们提交一个对 Graph.java 的改动（图中是把 Graph.java 的圆圈变成了星星），使得项目仓库的版本号变为了 2。如果我们接着改动 Trains.java 和 Node.java，项目仓库的仓库的版本号就会变成 3。关键是 Graph.java 也是在版本 3 中的，虽然它的内容相比版本 2 来说，并没有变过。

Subversion 的版本号并不是用来表明某个文件或者某些文件变动了多少[2]，因此不要尝试这么用。习惯于具体文件编号方式的朋友经常对自己在没有提交任何东西的情况下，项目仓库的版本往上增长了好多的现象困惑不

[2] 不管版本号是怎么分配的，使用它来了解“有多少改动发生过”是一点用处都没有的。因为一个改动可能是一个修改了文件每一行的改动。如果你想知道到底有多少改动，或许更好的方式是使用版本控制系统浏览历史的功能，直接去观察改动本身。

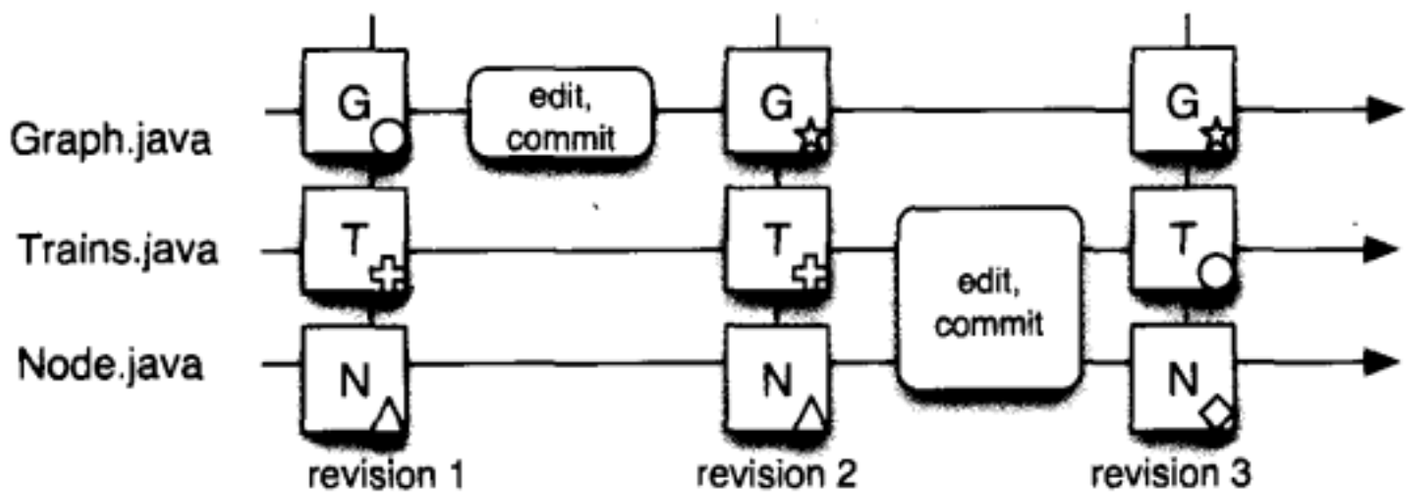


图 2.2 项目仓库的版本号

已。其实只要认识到不仅仅是你的改动，所有人的改动都会影响版本号，这个问题就很好理解了。

这样的存储历史版本的系统是相当强大的。有了它，版本控制系统可以做这样的一些事情：

- 获取文件的某个版本。
- 签出项目两个月前的源代码。
- 告诉你在版本7和9之间，某个文件的改动情况。

你还可以使用版本系统来撤销错误。如果你在一周快要结束的时候发现这周其实走到死胡同里去了，你可以备份好你所有做过的改动，然后把代码还原到星期一早上的模样。

2.6 标签

版本号的确是好东西，但是我们似乎更加喜欢记住像第二预览发布版这样的名字，而不是 r347 这样的数字。

Tags

标签是用来干这件事情的。版本控制系统让你可以给某一时刻的一组文件（或者一些目录或者整个项目）分配一个名字。如果你给我们的那三个文件分配了标签“第二发布预览版”，你以后就能使用这个标签签出它们了。

标签是一种很好地跟踪项目代码开发过程中发生重要事件的方式。我们会在本书中大量使用标签。你可以在第 115 页的第 9 章“使用标签和分支”中了解关于标签和分支（下一节的主题）的更多内容。

2.7 分支

在普通的开发过程中，大部分开发人员都在为同一份源代码添加内容（虽然他们可能在作的是不同的部分）。开发人员先签出代码，在他们的工作拷贝上作些修改，然后再签入回去，接着所有人都可以共享这次的工作成果。这样居于主要地位的那份源代码称之为“**主干**”。我们在图 2.3 中对此有图示。图中时间流逝是从左往右的（后面的图也是如此）。粗一些的水平线代表代码随着时间不断地被开发出来；这是开发的主线。开发人员从项目仓库中签出代码到他们自己的工作拷贝，然后从工作拷贝签入回项目仓库。

trunk

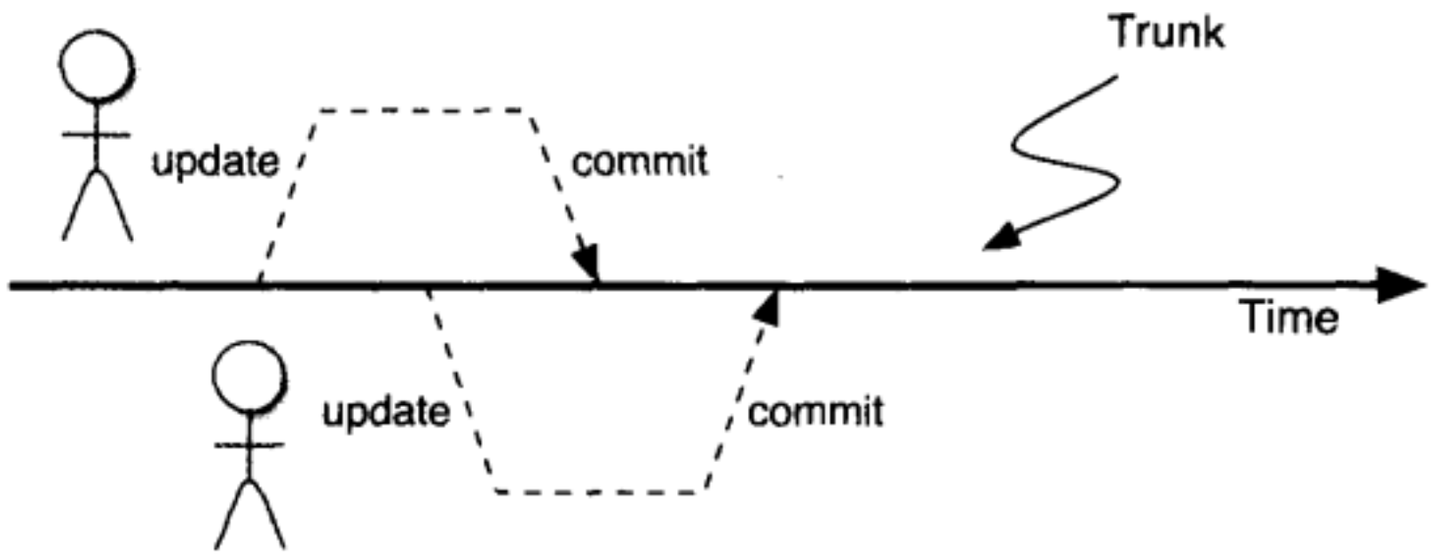


图 2.3 一个简单的主干

但是考虑一下这样一种情况，比如说快要发布一个新版本了。一个开发人员的小分队可能就要去为软件的这次发布做好准备，比如说修正一些收尾的 bug，与发布工程师一起工作，以及为 QA 团队提供帮助。在这个重要的时期，他们需要的是稳定性；如果这时还有其他的开发者编辑代码，添加一些预计到下次发布才会包括进去的功能特性的话，他们的努力都会白费。

一种选择是在要发布的时候停止当前的开发工作，但是这意味着团队的其他成员就要坐在那闲着没事干了。

另一种选择是把源代码拷贝到一台空闲的机器上，然后让发布团队使用这台机器工作。但是如果这样做的话，要是在拷贝出来后他们做了改动怎么办？我们又怎么跟踪那些改动呢？如果他们在要发布的代码中发现了 bug，而且这个 bug 也存在于主干之中，我们怎样做才能高效可靠地把修正代码合并回去呢？还有在他们发布之后，我们如何修正客户报告过来的 bug 呢？如何保证能找到发布时的那些源代码呢？

branching

一个比上面那些都要好得多的选择是使用版本控制系统中的“分支”功能。

分支有点像科幻故事中常见的那种设备，在这种设备中发生某种事件后会导致时间分裂。从那时往后，就会出现两个平行的将来。当另外某种事件发生时，又会导致这些未来再次分裂。很快，你就要面对大量平行的宇宙了（不得不说这是一种能够把你从故事编不出来的困境中解救出来的伟大设计）。

版本控制系统中的分支也可以让你创造多个平行的未来，但是其中既没有外星人，也没有星际牛仔，有的是源代码和版本信息。

接着说面临发布产品新版本的团队。到目前为止，整个团队都在“主干”（图 2.3 中的那条公共线）上工作。但是发布团队希望他们的工作能与主干隔离。为了达到这个目的，他们在项目仓库中创建了一个分支。从此刻直到他们的发布工作完成，发布团队都从这个分支签入签出东西。甚至当程序被发布之后，这个分支仍然是活动着的；如果客户报告了一些 bug，团队会在这个发布分支中修正它们。这在图 2.4 中有图示。

分支几乎就是一个完全独立的项目仓库：使用这个分支的人们可以看到它包含的代码并独立地进行操作，不会影响到工作在其他分支或者主干上的人。每个分支有自己的历史记录还能与主干独立地去跟踪改动（当然啦，分

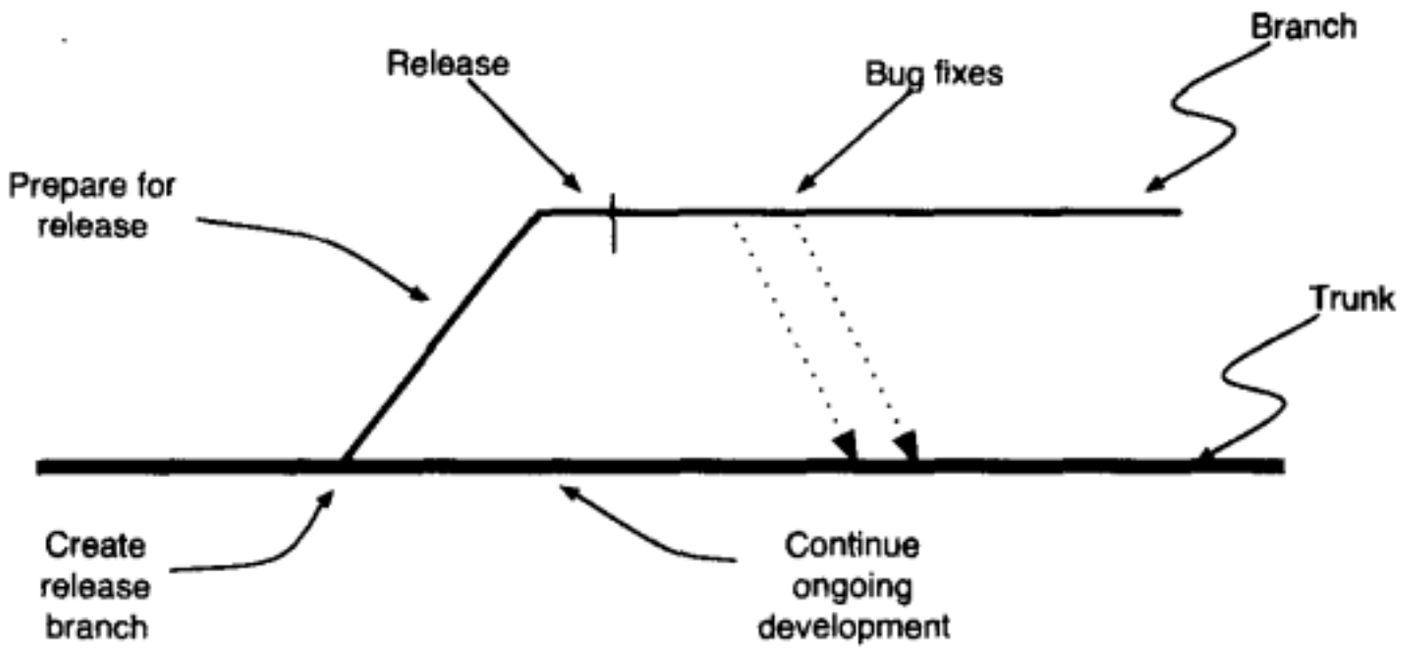


图 2.4 主干与发布分支

支出之前，分支和主干其实是一体的）。

这就是你在做发布的时候想要的东西。为发布而努力的团队会拥有稳定的代码，然后在此基础上润色，最后成功地发布。与此同时，大队伍的开发人员可以继续主线代码的开发；发布的时候再也不需要冻结代码了。而且当顾客报告了在发布版本中发现的问题时，团队可以从发布分支中找到代码，从而可以修正 bug 并发布更新过的版本，不用包含任何从主干那儿过来的新开发的代码。

分支在 Subversion 中是以命名目录的形式存储的；创建一个分支只须简单地把主干拷贝到一个新的位置。Subversion 内部的“**延迟拷贝**”实现方式使得拷贝过程很有效率，而且这样的延迟拷贝也是 Subversion 对代码打标签功能的基础。当你拷贝了一个文件或者目录时，Subversion 只是拷贝了指向原始文件的链接，当你对拷贝做了修改时，Subversion 把修改记录为相对原始文件的修改。使用延迟拷贝，Subversion 可以非常快地拷贝一大批文件而且几乎不占用空间，这对于分支和打标签来说很理想。

lazy copies

你可以在分支上创建新的分支，但是一般你无须这样做；我们遇到过不少开发人员发誓一辈子再也不用分支了，原因不过是因为在项目上有过过度使用分支的惨痛教训。

你应该避免过度的分支。虽然在开发时分支看起来是一种有利且廉价的做法，但是在你需要在不同分支间合并改动时是有高昂代价的。你不仅需要合并不同的代码行，还要确信你没有在这个过程中丢失任何做过的改动。记住当需要创建多个分支，特别分支是为了并行数个开发而不是做发布时，往往意味着有些事情不对了。

在本书中，我们会讲述一种简单的做事原则：只做必须要做的事情，回避无谓且复杂的事情。

2.8 合并

回到那个提到多个平行未来的科幻故事中。为了分开讲述，作者经常让他们的角色利用虫洞（polyphase deconfabulating oscillotrons）在不同的宇宙之间穿行，或者更简单点，道具干脆只是一杯冒着热气的茶。

在版本控制系统中，你也能在不同的未来中穿行（那杯茶就免了）。虽然每次签出的版本都来自于特定的分支，并且也是签入回同一个分支。但是在同一个开发人员的机器上签出数个分支也是容易做到的（当然是在硬盘的不同目录或者文件夹中）。这样做，开发人员可以在主干上写代码的同时给发布版本分支修复 bug。

merging

更妙的是，版本控制系统支持“合并”。比如说你在发布版本分支中修正了一个 bug 并且意识到同样的 bug 在主干代码中也存在。你可以让版本控制系统找出你为了修正这个 bug 在发布版本分支上做了哪些改动，然后把这些改动应用到主干代码上。你甚至可以把改动合并到数个不同发布版本的分支中。这很大程度上避免了把改动在系统的不同版本之间进行剪切拷贝。关于合并，后面我们还有很多东西要讨论。

2.9 加锁策略的选择

假设有两个开发人员，Fred 和 Wilma，他们在同一个项目中工作。他们两人都把项目的文件签出到了各自的本地硬盘上，并且都想要修改自己本地的 `File1.java`。如果这时两个人都签入会怎么样？

一个糟糕的情况可能是版本控制系统先接受了 Fred 做过的改动，然后又接受了这个文件的 Wilma 的版本。由于 Wilma 的版本中不可能会有 Fred 做过的改动，把 Wilma 的版本存进项目仓库的结果就是 Fred 在他硬盘上做过的改动因此丢失了。

为了避免这样的事情发生，版本控制系统必须实现某种解决冲突的系统（在 Fred 和 Wilma 这个例子中可能是一件好事情）。对于解决冲突，有两种常见的做法。

strict locking

第一种称为“**严格加锁**”。在以严格加锁方式工作的版本控制系统中，签出的所有文件一开始都被标记为“**只读**”。你可以阅读它们，用它们来构建你的系统，但是你不能编辑或者修改它们。要编辑或者修改，你必须征询项目仓库的同意：“我是否可以编辑 `File1.java`？”如果没有其他人在编辑同一个文件，那么项目仓库就会授权给你，与此同时把你本地的文件权限改为“可读写”。接着你就可以编辑了。如果其他人要编辑你正在编辑的文件，他们会被拒绝。在你完成了你的改动并把文件签入之后，你的本地文件又回到了只读的状态，同时该文件又能被其他人编辑了。

optimistic locking

解决冲突的第二种形式常被称作“**乐观加锁**”，虽然实际上它压根儿没有加锁。这种方式下，每个开发人员都能编辑所有签出的文件，也就是说，文件签出的时候处于可读写的状态。然而，项目仓库不会让你签入自你上次签出之后已经被更新的文件。它让你先更新本地的文件，包含项目仓库所有最新的改动之后再来签入。这就是其聪明之处了。版本控制系统不是简单地以项目仓库中最新版的文件去覆盖你辛辛苦苦改过的文件，而是尝试着去合并项目仓库中有的改动和你的改动。

例如，下面这个 File1.java：

```
Line 1  public class File1 {
    -     public String getName() {
    -       return "Wibble";
    -     }
    5     public int getSize() {
    -       return 42;
    -     }
    -   }
```

Wilma 和 Fred 都签出了这个文件。Fred 改动了第三行：

```
return "WIBBLE";
```

他然后又把文件签入了。这意味着 Wilma 的这个文件拷贝已经过期了。在不知情的情况下，Wilma 改动了第六行，让它返回 99 而不是 42。当她去把文件签入的时候，她被告知她的拷贝已经过期了；她需要合并项目仓库中已经有的改动。这与图 2.5 中标记“不同步”的星号是相对应的。

当 Wilma 把改动合并到她的文件中的时候，版本控制系统能够很聪明地发现 Fred 的改动并没有与她的重叠，因此它只是更新她的本地拷贝的第三行，把她的改动仍然留在她的文件中。当她签入的时候，她将只是存储她的改动而不会去碰 Fred 的部分。

要是 Fred 和 Wilma 改动的都是第三行，而且改动不一致会发生什么事情？假设 Fred 先签入，他的改动会被接受。当 Wilma 去签入的时候，她被告知她的拷贝过期了。这时，她去合并项目仓库中的版本，系统会注意到她改动了一个在项目仓库中已经改动过的地方。这是一个冲突。在这种情况下，Wilma 会看到一些警告信息，而且冲突会在她的那份源代码文件中标记出来。她必须手工地解决冲突（这时她可能会去询问 Fred 以找出他们同时修改同一行代码的原因）。

听我这么一说，你可能会认为乐观加锁是一种有些鲁莽的开发系统的方式：多个人同时编辑相同的一些文件。经常没有用过的人会臆测这样做不行，并且坚持只使用实现了严格加锁的版本控制系统。

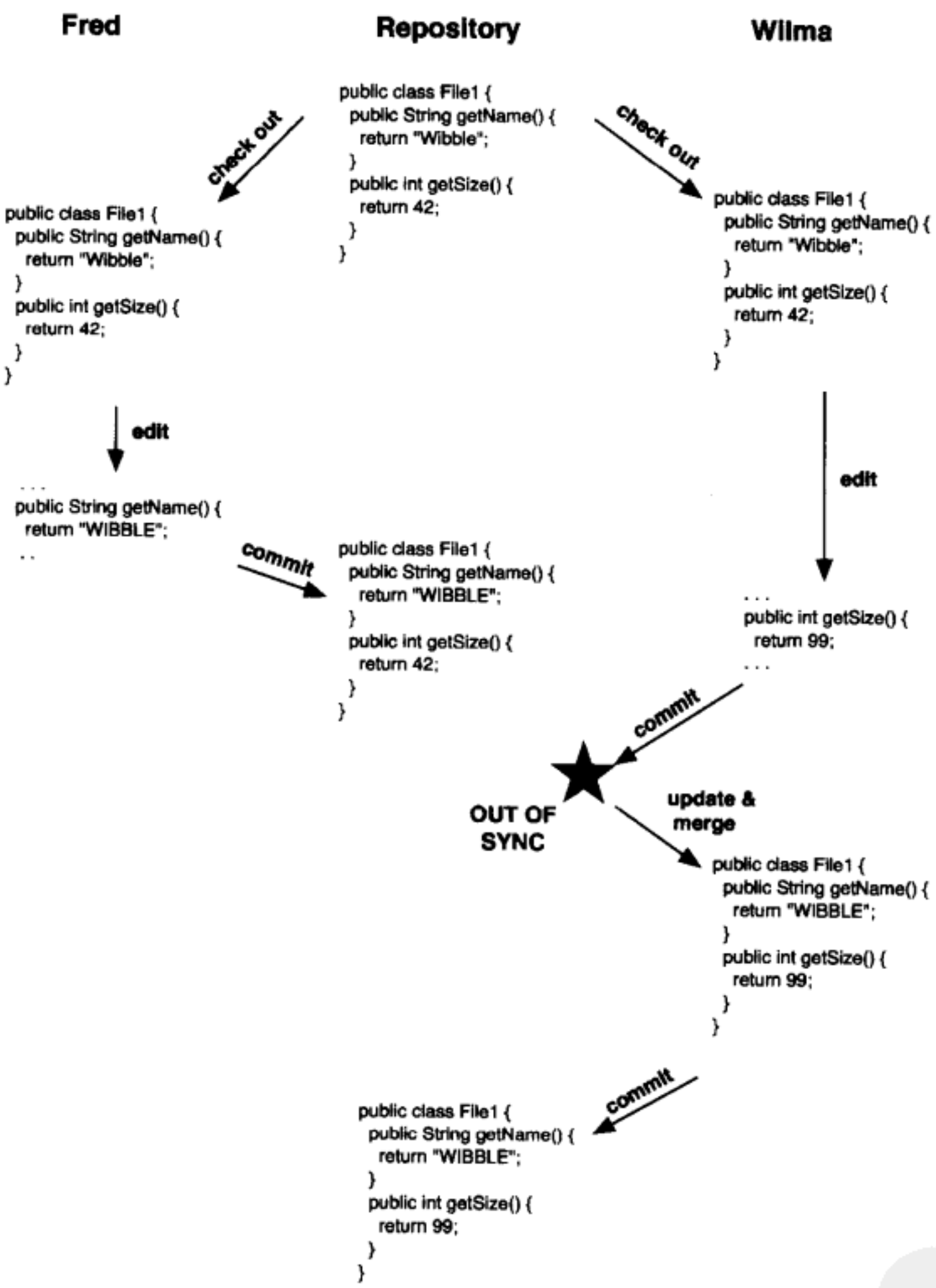


图 2.5 Fred 和 Wilma 改动了相同文件，但是冲突通过合并处理了

但是在现实生活中，严格加锁被实践证明总是给人带来很多烦恼并且没有什么特别的收益。如果你尝试一种乐观加锁的系统（像 Subversion），你会惊讶地发现冲突其实很少发生。事实证明在实践中团队内的划分工作的方式都会让人们工作在代码的不同区域中；他们只是偶然会遇见。而且当他们确实需要编辑同一个文件的时候，他们经常是编辑其不同部分。在严格加锁的系统中，一个人必须要等其他人结束编辑并签入了之后才能继续他的工作。在乐观加锁系统中，两个人都不会受影响。这么多年来，我们两种加锁方案都尝试过，而且我们强烈推荐团队的大部分人都应该使用乐观加锁的版本控制系统。

Subversion 1.2 引入了另一种可选的文件加锁方式，在第 103 页的第 7 章“文件加锁和二进制文件”中有讨论。只要简单地修改一个文件属性，就可以让 Subversion 对个别文件强制执行严格加锁。例如声音、图片或者其他无法合并的文件。

2.10 配置管理（CM）

有时你会听到人们谈论配置管理或者软件配置管理系统（或者使用缩写 CM 或者 SCM）。第一眼看过去，他们似乎是在谈论版本控制。而且这很大程度上是正确的；CM 的实践严重依赖于先有好的版本控制。但是版本控制只是配置管理使用的一个工具。

CM 是一套项目管理实践，它使得你能够精确可重现地交付软件。它使用版本控制来达到它的技术目标，但是它还使用了大量人工控制和交叉检查来保证事情没有被遗漏。你可以把配置管理想像成一种鉴别被交付的东西的做事方法，而版本控制是记录鉴别结果的手段。CM 是一个很庞大的议题，我们不会在本书中有更多涉及。如果你对 CM 感兴趣，《软件配置管理模式》

[BA03]是一个极好的资源，它在很多我们没法全面概括的问题上有深入细节的讨论。在此书中，许多技术和实例指导对应于一个 SCM 模式，我们会提到它们的名字。

但是对于现在，还是先集中精力让我们如何使用版本控制系统来完成我们的工作。下一章是对一个特定的版本控制系统 Subversion 的简单介绍。

PDG

第 3 章

Subversion 入门

Getting Started with Subversion

熟悉一个新的工具软件的最好方式就是动手去尝试，因此，本章将向你展示如何创建一个真实的 Subversion 项目仓库并在其上做开发。你将会学到使用 Subversion 的基本步骤以及如何维护一个简单的项目。

因为 Subversion 相对较新，所以可能需要在你的机器上安装它才能用。本章要讲的基本安装是相当简单的。要了解更高级的安装方式、联网、安全和管理方面的内容，参见第 157 页的附录 A。

Subversion 自带了一个命令行的客户端，但是有很多第三方工具可以与你的项目仓库交互。比如与 Windows Explorer 集成的 TortoiseSVN，以及一些支持 Subversion 的 IDE。

3.1 安装 Subversion

显然你需要先安装 Subversion 才能使用。你可以把 Subversion 的客户端和服务器端分开来安装，这取决于 Subversion 在你的操作系统上是如何打包的。这样的选择对于 Unix 平台来说更常见，在其上管理员可能想要设置一个没有安装客户端工具的服务器。

Joe 问……

Shell，提示符，命令行窗口

当你要使用命令行的时候，相关术语可能会很让人搞不清楚方向，所以让我们先把这个讲清楚些。

命令处理器，也称之为“*shell*”，是一个接受并且执行命令的程序。命令可以带有参数，而且命令处理器经常有额外的功能（例如重定向程序的输出到一个文件）。在 Windows 下，`cmd` 和 `command` 是常见的命令处理器（使用哪个取决于你的 Windows 版本）。在 Unix 主机上，有更多的 shell 供选择，从最初的 `sh` 到 `csh`、`bash`、`tcsh`、`zsh` 等等。

在我们有 GUI 系统之前，命令处理器或者 shell 就是我们与计算机交互的方式。当你启动 DOS 看见了 DOS 的提示符时，这时就是在与命令程序交互了；你的计算机显示器只是一个终端而已。

今天我们已经有了很酷的前端了，我们需要一个地方来运行这些命令处理器，因此人们写了一些命令行程序运行在 Windows 中。当这样的终端应用程序在运行一个命令处理器或者 shell 的时候，你可以在提示符处输入命令并且执行它们。有时我们会称这些执行命令的窗口为**命令窗口**。

我们的第一步是检查 Subversion 是否已经安装到你的电脑上了。最简单的方式是使用命令行。如果你熟悉命令行，你可以略过下一节。

■ 命令行

命令行是一个底层设施，它让你可以直接在你的电脑上运行命令。命令行是一个强大的工具，但是它也是很晦涩难懂的：你发出命令的时候，你就像弯着腰在引擎室内工作一样。

在 Windows 主机上，你可以通过使用“**开始>运行**”并输入 `cmd` 作为要运行的程序名来打开一个命令行窗口（在老一些的 Windows 版本上，你可能需要输入的是 `command` 而不是 `cmd`）。你应该看到像图 3.1 那样的窗口。

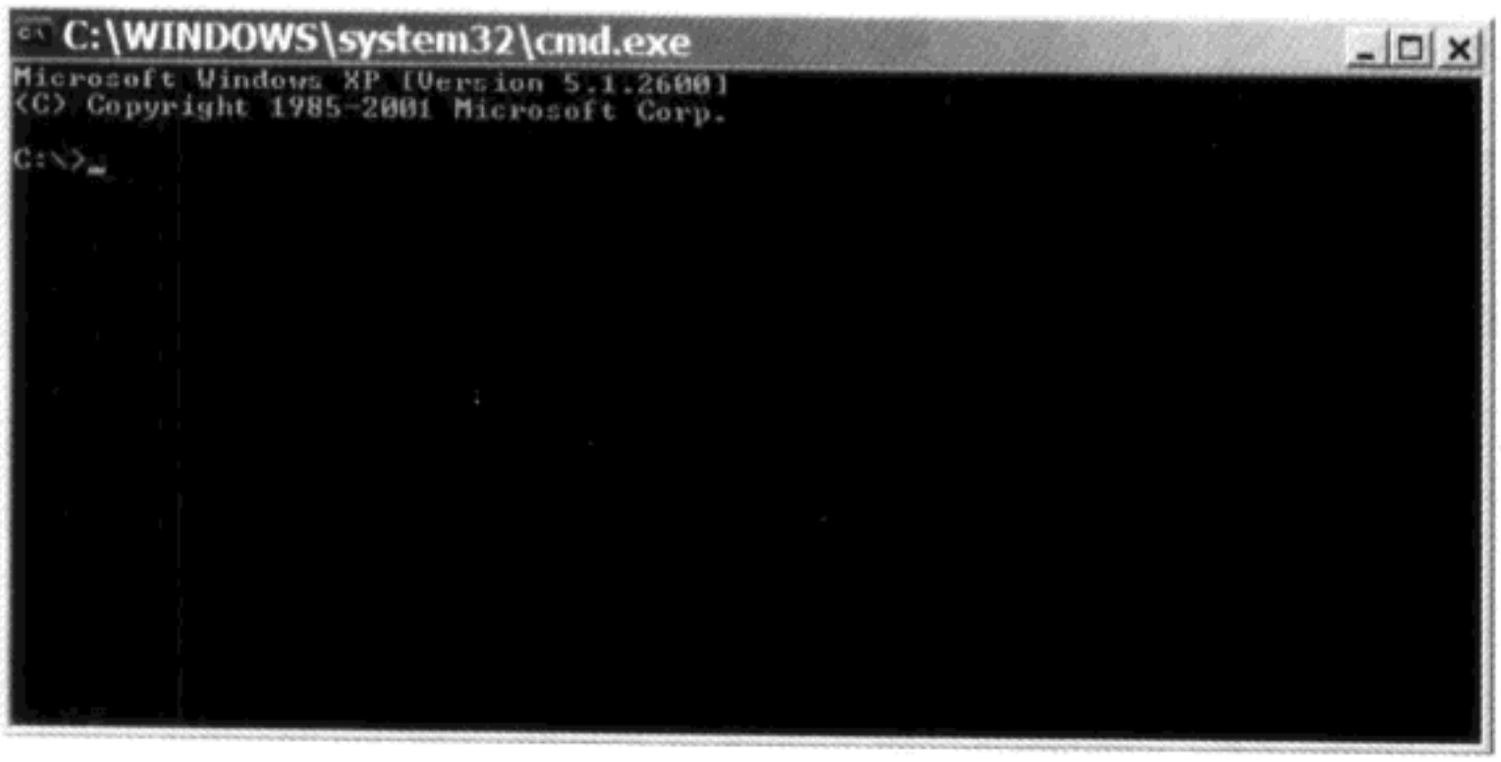


图 3.1 Windows 命令提示符

在 Unix 主机上，你也许已经在命令行上工作了。如果不是这样，你使用的是像 Gnome 或者 KDE 这样的桌面环境，去寻找 `terminal`、`konsole` 或者 `xterm` 这样的程序，运行它们。你应该可以看到一个类似图 3.2 所示的窗口。如果你正在使用的是 Mac OS X，你的 shell 程序藏在目录 `/Applications/Utilities/Terminal` 中。

你使用命令行窗口来输入命令并查看它们的输出；在这里没有 GUI 前端。

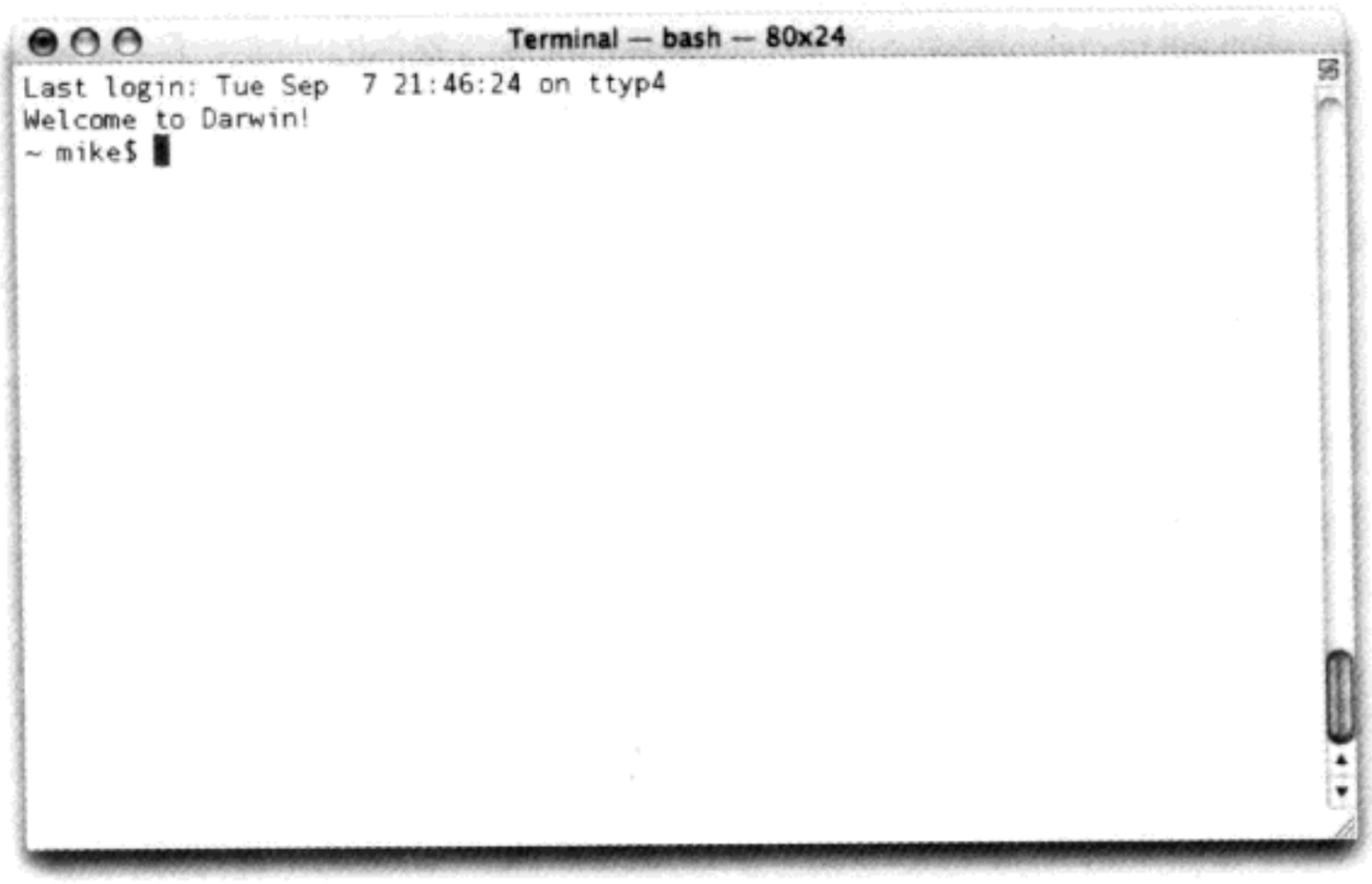


图 3.2 Unix Shell 提示符

例如，在你刚创建的命令行窗口中，输入下面的命令并按回车键（有时这个按键被标做 Return）：

```
echo Hello
```

你应该看见文字“Hello”被返回给你了，而且在那之下又出现了一个新的提示符，在那儿你可以输入另外一个命令。图 3.3 中有一个例子。

■ 提示符

命令窗口的一个乐趣是可以定制 shell 的提示符，是用来告诉你它已经准备好接收输入的符号。你可以在提示符中包括时间、当前目录、你的用户名以及其他各种重要信息。不幸的是，灵活的同时也让人迷惑：看前面那张截图你会发现 Windows 提示符看起来和 Unix 提示符是不同的。

在本书中，我们在例子中尝试着通过标准化通用提示符来简化事情。我们会显示当前目录的名字后跟大于号（>）。例如，下面这样的命令：

```
work> svn update
```

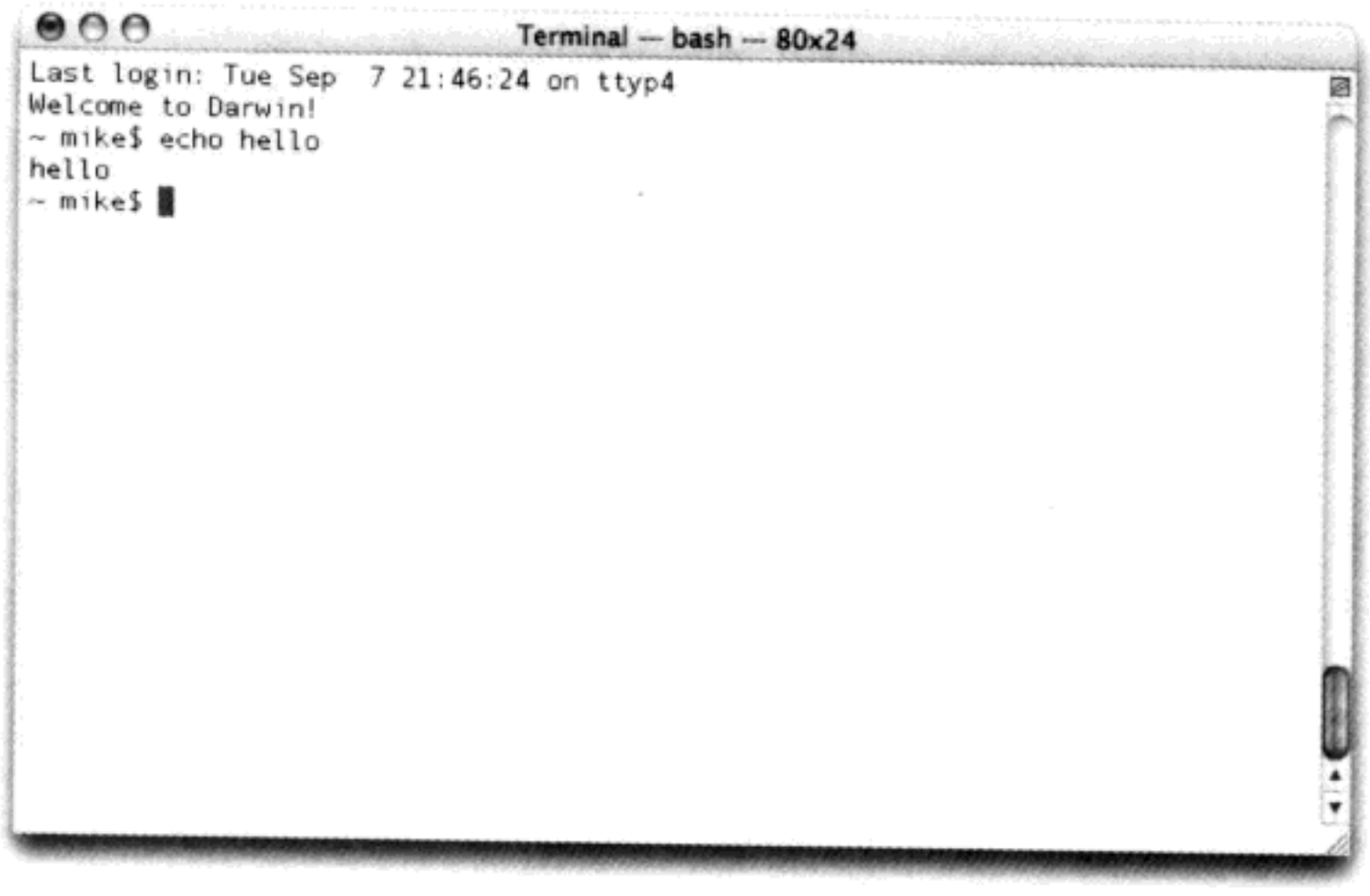


图 3.3 “hello”返回之后

这意味着我们在一个叫做 work 的目录中发出了一个叫做 svn update 的命令。把这个“逻辑上的”命令映射到命令窗口中去应该是很简单的。

本书中的命令一般都不是 Windows 或者 Unix 特有的：它们应该在两个系统上都能用。唯一不同的地方是文件名；Windows 使用盘符以及反斜杠来区分文件名中的各个部分，而 Unix 使用的是斜杠。根据你的环境，选择合适的文件名就应该没问题。这个规则的一个例外是基于 file://的项目仓库，Windows 和 Unix 的语法稍微有一点不同。在这种情况下，我们会既讲 Windows 版本的命令，也讲 Unix 版本的命令。

■ 检查是否安装了 Subversion

在你的电脑上打开命令窗口，并输入命令 svn-version，然后按回车。如果 Subversion 客户端被正确安装了，你应该可以看到一个类似于图 3.4 那

样的回应。接着尝试 svnadmin-version，看看 Subversion 的管理工具有没有被安装。如果两个命令都能用，你可以直接跳到下一节了。

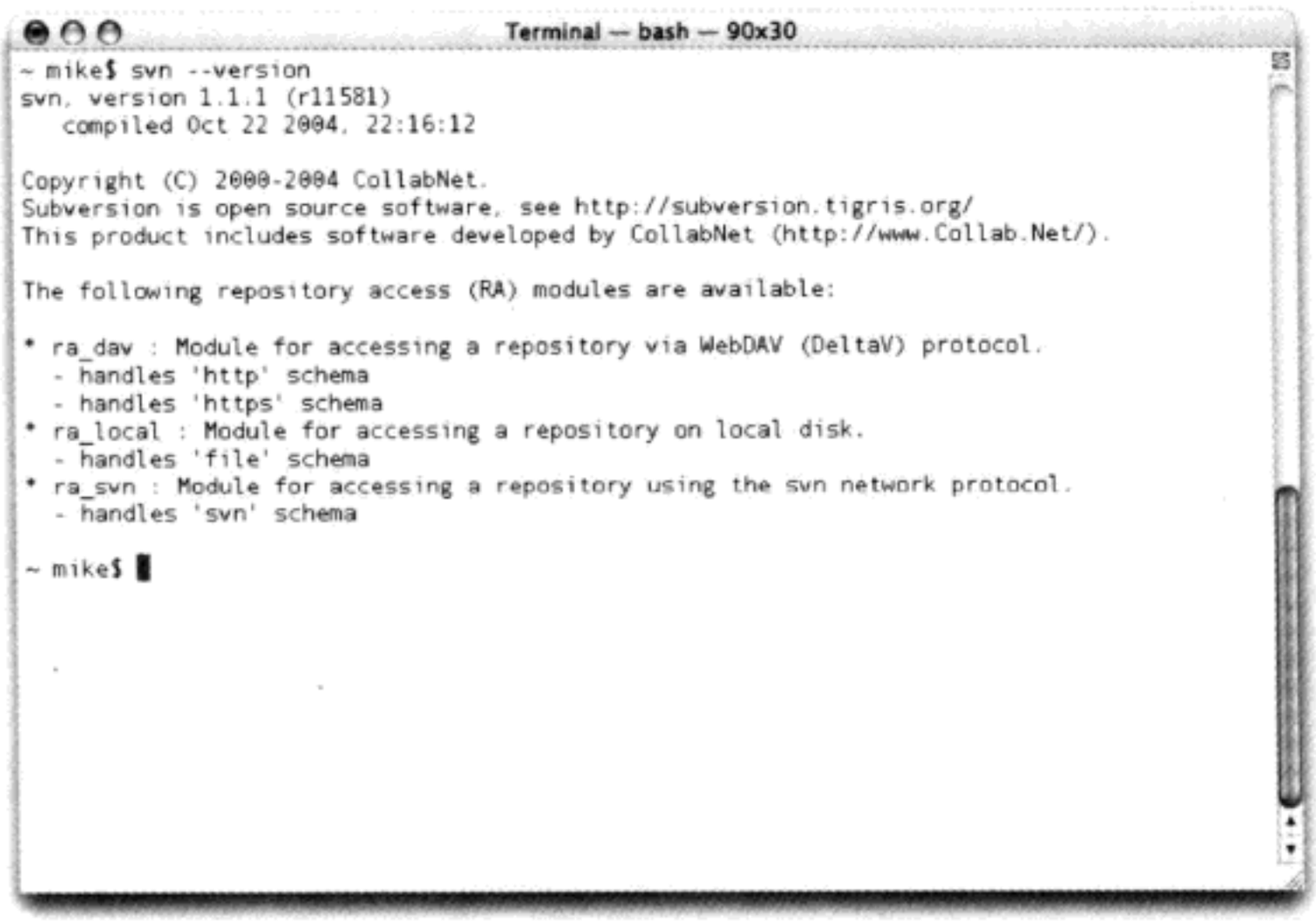


图 3.4 正确安装的 Subversion 客户端

最有可能的情况是你的电脑告诉你，它找不到 svn 或者 svnadmin。这没关系，Subversion 还不是大部分操作系统标准安装的一部分。Subversion 在不同的操作系统上都同时有源代码和二进制包发布。在下载页面 http://subversion.tigris.org/project_packages.html 中，应该有针对你的操作系统的完整的安装指导。如果你想要自己编译 Subversion 的话，你也可以下载源代码，但是因为 Subversion 依赖于大量其他的包，直接下载编译好的版本可能是最简单的办法。

3.2 创建项目仓库

Subversion 需要项目仓库来储存你的数据。在本节中，你将创建一个项目仓库来储存你的第一个项目。

Subversion 的版本

Subversion 的开发者是很勤奋的，自从本书第 1 版以来，Subversion 发布了 1.2 和 1.3 版。本书中的大部分例子应该能在 Subversion 的任何版本上运行，但是文件加锁特性需要 Subversion 1.2（或者更高）而更高级的身份验证功能需要 Subversion 1.3（或者更高）。

当一个特性需要特定版本的 Subversion 时，会有一个标志来提示你的。如果可能，我们一般推荐使用 Subversion 的最新版本，因为它会更稳定，能得到的支持也更多。

首先你需要给项目仓库创建一个空白目录，然后告诉 Subversion 在这个目录中创建一个新的项目仓库。让我们假定你使用的目录是 `/home/mike/svn-repos`（对 Unix 而言），或者 `c:\svn-repos`（对 Windows 而言）。

Windows:

```
mkdir c:\svn-repos
svnadmin create c:\svn-repos
```

Unix:

```
mkdir /home/mike/svn-repos
svnadmin create /home/mike/svn-repos
```

输入 `svnadmin` 命令之后，在你的项目仓库目录中应该就有一些文件了。后面我们会详细讲解项目仓库是如何储存在硬盘上的，但是现在你只要把项目仓库和其中的内容当作黑盒就可以了。

这样你的 Subversion 项目仓库就设置好了，接着我们就开始创建项目。

3.3 创建简单的项目

让我们在你的项目仓库中添加一个新的项目。使用一个不知所以但是听起来很酷的项目名——*Sesame*。好的，我们从创建一些文件然后把它们导

使用远程文件系统

如果你使用的是远程文件系统，例如 Windows 的个人目录是在网络上共享的或者你的 Unix 个人目录是由 NFS 映射而来的，Subversion 的客户端都能很好地工作。你可以把工作拷贝签出到任意网络驱动器上，没有问题。

如果你运行的是 Subversion“服务器”，那就要小心一点了。Subversion 1.0 随同发布的有 Berkeley DB，用它来做存放项目仓库的后端。BDB 不喜欢放在网络驱动器上的数据库文件，因为它要把它们映射到内存中。

Subversion 1.1 引入了基于文件系统的“fsfs”，这个在 Subversion 1.2 之后变成默认选项了。如果你使用的是 Subversion 1.2 或者 1.3，使用 `svnadmin create` 创建的项目仓库可以在远程文件系统上正常工作。

如果你想要使用 BDB 后端来代替 fsfs，在创建你的项目仓库时指定 `--fs-type bdb` 选项。当使用 BDB 时，你必须把项目仓库存储在本地驱动器上。

入项目仓库的 `sesame` 目录开始吧（项目名正式的说法是 Sesame，但是我们会在项目仓库中使用小写的 *sesame* ）。

在你的电脑上创建一个叫做 `tmpdir` 的临时目录。在这个目录中，使用你最顺手的文本编辑器来创建两个文件：`Day.txt` 和 `Number.txt`。

文件 `Day.txt`：

```
monday
tuesday
wednesday
thursday
friday
```

文件 `Number.txt`：

```
zero
one
two
three
four
```

这些东西看起来不大像程序源代码的样子，但是记住我们使用项目仓库存储所有用来构建项目的东西。这么看来 Sesame 好像须要知道一个星期内每天的名字以及一些小的数字，而这些就是所需的数据文件。

我们现在须要让 Subversion 把这些文件导入到项目仓库中的一个新项目中去。Subversion 通过目录组织所有东西，在第 111 页的第 8 章“组织你的项目仓库”中有更深入的讨论。就现在而言，我们只要按照 Subversion 开发人员推荐的习惯用法把 Sesame 项目储存在/sesame/trunk 中就可以了。

在你的命令行中，把当前目录切换到 tmpdir 目录。如果你在 Windows 上运行：

```
tmpdir> svn import -m "importing Sesame project" \
              . file:///c:/svn-repos/sesame/trunk
Adding          Number.txt
Adding          Day.txt
Committed revision 1.
```

不要输入日志消息后面的反斜杠。我们使用 \ 来把一行分割成多行，是因为一行写不下了。在本书中你会见到很多这样的情况。

如果你使用的是 Unix，运行：

```
tmpdir> svn import -m "importing Sesame project" \
              . file:///home/mike/svn-repos/sesame/trunk
Adding          Number.txt
Adding          Day.txt
Committed revision 1.
```

import 关键字告诉 Subversion，我们想做的是把一些文件导入到项目仓库之中。-m 选项使得你可以给这次导入操作关联一条消息。使用日志消息来表明你执行的导入操作的意图是不错的主意。

下一个参数(.)告诉 Subversion 导入当前目录也就是 tmpdir 的内容到项目仓库之中。最后一个参数是项目仓库的 URL，描述了我们要把文件导入到哪儿去。这儿我们说的是 Subversion 去本地文件系统中寻找位于 svn-repos 目录中的项目仓库，然后导入到其中的/sesame/trunk。[1]

[1] 我们导入到/sesame/trunk 中是因为在将来 Sesame 项目须要支持分支。这些分支会储存在/sesame/branches 中。这会在第 111 页的第 8 章“组织你的项目仓库”中有更完整的讨论。

项目仓库 URL

你可能已经注意到了当我们把文件导入到项目仓库中时，使用了 `file:///...`URL 来告诉 Subversion 把新项目放到哪儿。合格语法看起来很像你在 web 浏览器中看到的互联网上的地址，除了 URL 是以 `file:///` 开头而不是`http://`。这告诉 Subversion 到本地文件系统中寻找项目仓库，而不是去 web 上找。

在第 57 页的第 5 章“访问项目仓库”中，你将看到如何通过网络使用不同的 URL 来访问 Subversion 的项目仓库。这些项目仓库要么在 web 服务器上，要么须通过自定义的 `svn` 协议来访问。

Subversion 回应说它已经添加了两个文件并且把改动提交到项目仓库中了。

现在已经把这些文件安全地放在项目仓库中了。鲁莽（或者说愚蠢）的开发等，这时可能会跑到临时目录中把这些拷贝给删掉。然而，谨慎（而且务实）的开发者可能先要验证它们确实已经储存在项目仓库中，然后才会把它们删掉。而要知道是否已经存储进去的最简单的办法就是让 Subversion 把在 Sesame 项目中的文件签出到你的本地工作目录。一旦我们确认所有东西都在那儿，而且看起来没啥问题，就可以把原来的那份给删除了。下一节就是讲如何来实际做这些事情的。

3.4 开始开发一个项目

无论你是重新开始开发一个新项目（例如我们刚创建的 Sesame），还是刚加入一个已经开发了几个月并已经有了数千源文件的项目，这都没关系。你要上手去开发项目所要做的都是一样的：

1. 决定把文件的工作拷贝放在你本地机器什么位置。

2. 把项目签出到什么位置。

第一个问题一般都是很容易决定的。我们一般习惯在我们的机器上建立一个叫做 work 的目录。然后把所有的项目都签出到那个目录之下。对于简单的项目，我们一般就直接把文件签出到 work 之下。对于更复杂的项目，可能要包括数个代码分支，所以我们会把东西整理到几个子目录下。就现在而言，假定我们开发的项目是简单项目。如果我们签出了三个不同的项目，分别叫做 poppy、sesame 和 sunflower，最后我们的目录看起来就像图 3.5 中画的那样。

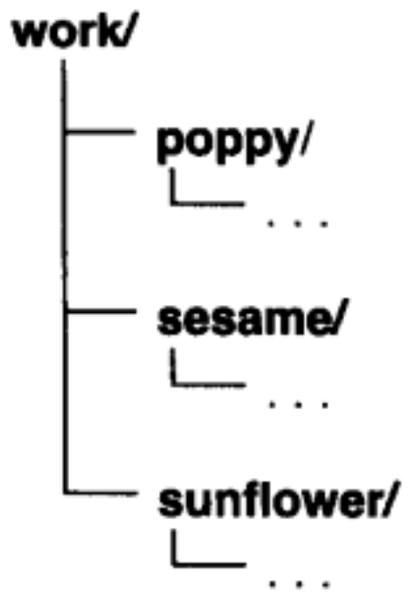


图 3.5 工作目录的结构

如果你没有 work 目录的话，那就先创建一个。创建目录既可以使用命令行，也可以使用你的文件管理器。

Windows: `mkdir c:\work`

Unix: `mkdir /home/mike/work`

现在要把源代码签出到工作目录中。我们使用 file://URL 来指定项目仓库，这个命令在 Windows 和 Unix 下又不一样了。

把当前目录切换到 work 目录，在 Windows 上可运行：

```
work> svn co file:///c:/svn-repos/sesame/trunk sesame
A   sesame\Number.txt
A   sesame\Day.txt
Checked out revision 1.
```

在 Unix 上，你须要运行：

```
work> svn co file:///home/mike/svn-repos/sesame/trunk sesame
A   sesame/Number.txt
A   sesame/Day.txt
Checked out revision 1.
```

参数 co 告诉 Subversion 想要执行签出操作，而 file://URL 指定了想要签出的项目仓库的位置，最后告诉 Subversion 想要把工作拷贝签出到哪儿，在本例中是签出到工作目录中的 sesame 目录。

你现在有了 Sesame 项目的一个本地拷贝[2]了，它包含了最初导入的两个文件。从现在开始，就要一直在文件的这个拷贝上工作了，因为它们是被 Subversion 管理着的。检查过它们看起来没问题之后，可以去临时目录把原始拷贝给删除了。我们已经把这些文件的控制权交给了版本控制系统，要是在机器中同时存放有原始版本和被 Subversion 管理着的版本的话，就太让人迷惑了。我们把 sesame 作为当前目录，并且在签出的文件上做开发。

3.5 修改

尽管我们工作很努力，但是客户还是跑来抱怨：软件应该支持周末。好的，打开你最顺手的编辑器在 Day.txt 后面添加两行：

```
monday
tuesday
wednesday
thursday
friday
saturday
sunday
```

[2] Subversion 把这些称作项目仓库文件的“工作拷贝”，而这对应于 SCM 的“private workspace”模式。

在把这些改动存回硬盘之后，让我们看看 Subversion 认为项目的状态是怎样的吧。你可以使用 `svn status` 命令来获得一个或者多个文件的状态：

```
sesame> svn status Day.txt
M       Day.txt
```

M 在这儿表示 Subversion 认出这个文件已经在本地被修改过了（而且这些改动还没有被储存到项目仓库中去）。

如果我们把工作都分为一小步一小步地去做，要记住给三两个文件做了什么改动不是难事。然而，如果你忘记了文件为什么被修改了（或者你只是想要再检查一下），可以使用 `svn diff` 命令来显示文件在项目仓库中的版本和本地拷贝之间的区别：

```
sesame> svn diff Day.txt
Index: Day.txt
===================================================================
--- Day.txt (revision 1)
+++ Day.txt (working copy)
@@ -3,3 +3,5 @@
  wednesday
  thursday
  friday
+saturday
+sunday
```

输出中包含着大量信息。第一行告诉正在检查的文件的名字。这有好几个用处。首先，如果用一条命令检查数个文件，它帮助我们辨认出读到哪儿了。其次，它还用来生成补丁（但是现在还不用去管这些）。

等号那行之后的两行告诉我们项目仓库文件的名字和版本号，可以看出正在把它与工作拷贝进行比对。

`@@ -3,3 +3,5 @@`有些不好理解，它告诉我们修改过的地方在文件的哪个位置，跟着的是实际的改动。以 + 号开头的行表示它们是新添加的行，而以 - 号开头的行表示这些行被移除了。

改动是以“统一”的格式表示的，意味着它既包含了上下文信息，也包含了被改动的行。这是一个流行的格式，因为它很容易阅读，而且额外的上

下文信息使得它可以在原始文件被稍微改动了一些后还能被应用。Subversion 还允许使用--diff-cmd 命令来指定自己希望使用的查找改动的程序。这很有用，比如我们想要使用图形界面工具时。

在这种场合，使用 GUI 前端有着无与伦比的优势：如果你使用这样的工具，你可以产生用颜色标记得很美观的报表来显示文件之间的差异。

除此之外，Subversion 的 diff 命令可以显示你的工作拷贝和项目仓库中指定版本之间，或者项目仓库中两个不同版本之间的差异。第 83 页的 6.6 节“使用 Subversion 版本标识符”对于 diff 选项有更详细的讨论。

3.6 更新项目仓库

做完了修改（当然单元测试也运行过了），就可以把最新的版本储存到项目仓库之中去了。在像 Sesame 这样一个人做的项目中，这是非常简单的事情。使用 svn commit 命令：

```
sesame> svn commit -m "Client wants us to work on weekends"
Sending        Day.txt
Transmitting file data .
Committed revision 2.
```

commit 的功能是储存我们做的所有改动到项目仓库中。-m 选项是用来给所做的改动附加有意义的注释用的。

虽然我们让 Subversion 提交 Sesame 项目中的所有文件，它还是能很聪明地发现 Number.txt 并没有被修改，因此只会把 Day.txt 中的改动发送给项目仓库。

Subversion 告诉我们它“提交了版本 2。”值得注意的是，版本号 2 是针对整个项目仓库而言，而不仅仅是 Day.txt。如果我们既改动了 Day.txt 也改动了 Number.txt，得到的版本号仍然是项目仓库的版本 2。你可以把 Subversion 的版本号看做某种全局的标记，伴随着项目仓库一直往上升，记录着每一次签入的改动。

在提交之后，可以使用日志功能来确认项目仓库确实被更新了：

```
sesame> svn log Day.txt
------------------------------------------------------------
r2 | mike | 2004-09-08 21:54:19 -0600 (Wed, 08 Sep 2004)
Client wants us to work on weekends
------------------------------------------------------------
r1 | mike | 2004-09-08 21:50:13 -0600 (Wed, 08 Sep 2004)
importing Sesame project
------------------------------------------------------------
```

我们可以看到 mike 是最后一个改动过 Day.txt 的用户，他改动的是项目仓库的版本号 2，而且我们看到在添加星期六和星期日到日期列表时使用的日志消息。还可以看到 Day.txt 在版本 1 中就被改动了，那时我们做的是把 Sesame 项目给导入了。如果你使用--verbose，Subversion 会告诉你每个版本中具体有哪些被改动了：

```
sesame> svn log --verbose Day.txt
------------------------------------------------------------
r2 | mike | 2004-09-08 21:54:19 -0600 (Wed, 08 Sep 2004)
Changed paths:
   M /sesame/trunk/Day.txt
Client wants us to work on weekends
------------------------------------------------------------
r1 | mike | 2004-09-08 21:50:13 -0600 (Wed, 08 Sep 2004)
Changed paths:
   A /sesame
   A /sesame/trunk
   A /sesame/trunk/Day.txt
   A /sesame/trunk/Number.txt
importing Sesame project
------------------------------------------------------------
```

现在 Subversion 的话变得更多了，我们可以看到在版本 2 中 /sesame/trunk/Day.txt 被改动了，靠着它有一个 M 标志。对于版本 1，我们可以看到/sesame 目录和它的内容被创建了。因为 Subversion 是这样跟踪提交（每次提交就是对一些文件的改动）的：所有的改动都是一次存起来，然后关联一个日志消息。它能显示每次提交中所有改动过的文件，即便我们询问的只是 Day.txt。这是很有用的，比如在修复一个 bug 要查找历史信息的时候。

■ 混合版本的工作拷贝

在最后一个例子中，我们使用了 svn log 来查看 Day.txt 的历史。事实上，使用 svn log 不加任何其他参数产生的是当前目录以及其所有子目录的日志信息，从最近的改动开始往后列开去。

设置消息编辑器

无论你何时改动项目仓库，通过导入文件也好，提交改动也好，或者在其中拷贝一些东西来去也好，你都需要输入日志消息。如果你不指定-m选项，Subversion 会尝试着打开一个编辑器让你输入日志消息。

Subversion 在环境变量中查找 SVN_EDITOR、VISUAL 或者 EDITOR 来决定它应该使用的编辑器。如果你是在 Windows 平台上，而且想要把编辑器设置为记事本，打开命令行输入：

```
work> set SVN_EDITOR=notepad
```

这样设置的 SVN_EDITOR 只会对当前命令窗口有效。如果你想要永久设置环境变量，你要到 Windows 的控制面板（如果使用的是 Windows XP 还要切换到经典视图）中选择“系统”。在“高级”选项页，点击“环境变量”按钮，然后创建一个新的变量。变量名应该是 SVN_EDITOR，值应该是 notepad。在设置好了新的环境变量之后，需要关闭所有已经打开的命令窗口然后重新打开它们以让新的设置生效。

如果你是 Unix 用户，需要根据你所使用的 shell 设置对应的环境变量。尝试在你个人目录的.profile、.bashrc 或者.cshrc 这些文件中寻找已经存在的环境变量，然后添加一个新的。你可能需要登出之后再登录一遍以让设置生效。

如果你在提交了对 Day.txt 所作的改动之后立马去询问当前目录的日志，Subversion 是不会告诉你有什么改动的。这有一点违反直觉，不是吗？毕竟所有的改动都已经在项目仓库中了，我们询问 Day.txt 的日志的话，应该是可以看得到的呀。那为什么 Subversion 没有把改动写进当前目录的日志中去呢？

答案是因为 Subversion 把目录当作一等公民来跟踪。它在你的工作拷贝中记住了每个目录的版本号。当我们提交了 Day.txt 的改动之后，Subversion 知道了工作拷贝的版本号是 2，但是实际上目录仍然是版本 1 的目录。为了能够看到日志信息，你要首先运行 svn update，把当前目录更新为版本 2。

大部分时间可以忽略混合版本的问题。如果你确实给这样的行为困住了，svn update 一下就能解决这个问题。在本书后面的实例指导中，我们经常把更新作为第一步要做的事情，以避免这样的问题出现。

3.7 当世界产生碰撞时

每个人在第一次听到 Subversion 不会因为编辑而把文件锁住时，都会紧张。他们会想“要是两个人同时在编辑同一个文件会发生什么？”。在本节中我们会提供答案（而且幸运的话这个过程还能消解你可能有的忧虑）。为了制造这样的情况，我们还需要另外一个用户（为的是可以有多个人同时编辑同一个文件）。不幸的是，我们的人体克隆机器还在运输途中，所以我们只能让你自己模拟出另外一个自己了。

在处理冲突的时候，Subversion 其实并不知道用户这个概念。相反，它关心的是不同的工作拷贝是不是与项目仓库一致。这意味着我们可以简单地通过新签出项目的另外一份拷贝，来达到模拟另外一个用户的目的；我们要做的不过是把它放在与第一份拷贝不同的位置而已。当第一次签出我们的项目时，把它放在与项目同名的叫做 sesame 的目录中。为了再签出一次，需

要指定另外一个地方，它的目录应该与正在开发的目录平行。让我们把这个目录称作 aladdin 吧。在 Windows 上，再次签出需要这么做：

```
work> svn co file:///c:/svn-repos/sesame/trunk aladdin
A   aladdin\Number.txt
A   aladdin\Day.txt
Checked out revision 2.
```

在 Unix 系统上，你需要运行：

```
work> svn co file:///home/mike/svn-repos/sesame/trunk aladdin
A   aladdin/Number.txt
A   aladdin/Day.txt
Checked out revision 2.
```

我们已经从同一个项目仓库中签出了一个项目（Sesame），已经都在其上做过开发了。这次要把文件存储到一个称作 aladdin 的新目录。因为原来的目录文件都签入了，现在在硬盘上就有了两份项目的拷贝，一份位于 sesame，另外一份在 aladdin 中。目前两份是一模一样的（不信的话，读者尽管去检查）。有两个不同的目录，是因为我们要模拟两个人同时开发一个项目的场景，每个人都有自己签出的文件拷贝。

让我们首先来快速检查一下 Subversion 的神智是否清醒。我们先改变一个目录，签入，然后让 Subversion 更新另外一个目录中的本地拷贝。

首先，编辑 sesame 目录中的 Number.txt，添加两个新行（五和六）：

```
zero
one
two
three
four
five
six
```

现在，把文件签入项目仓库：

```
sesame> svn commit -m "Customer wants more numbers"
Sending        Number.txt
Transmitting file data .
Committed revision 3.
```

现在来揭开真相。在 aladdin 目录中，该版本中的 Number.txt 现在是过期了的（因为项目仓库中有一个更新的版本）。让我们检查一下：

```
sesame> cd ..
work> cd aladdin
aladdin> svn status --show-updates
       *          2   Number.txt
Status against revision:      3
```

我们使用--show-updates（简写为-u）去让 Subversion 从项目仓库中找出 aladdin 目录中的那些文件是否有更新。我们需要使用这个选项，是因为默认配置下 Subversion 只是检查文件在工作拷贝中是否有本地改动，而不是去检查在项目仓库是否有一个更新的版本可用。

星号表示位于版本 2 的 Number.txt 有一项更新。在执行这个检查的时候，Subversion 还告诉我们项目仓库当前版本号是 3。

在我们更新到最新版本之前，可能要问 Subversion，更新版本和项目仓库中当前的版本之间的差异是什么（因为如果更新它，会影响到当你正在开发东西的时候，你就须要过段时间再更新）。我们再一次使用 svn diff 命令：

```
aladdin> svn diff -rHEAD Number.txt
Index: Number.txt
===================================================================
--- Number.txt (revision 3)
+++ Number.txt (working copy)
@@ -3,5 +3,3 @@
  two
  three
  four
-five
-six
```

-rHEAD 选项告诉 Subversion，我们须比较 Number.txt 的本地拷贝与项目仓库中最新版本之间的差别。又一次见到了那令人费解的@@ -3,5 +3,3 @@。可以看出在工作拷贝中没有那新添加的两行（当然这并不令人惊讶）。如果我们没有指定-r 标志，Subversion 会把本地拷贝的 Number.txt 与项目仓库的签出版本（在本例中是 r2）拿来进行比较。因为 Aladdin 还没有修改文件，这样会显示没有改动。

我们可以更新在 aladdin 目录中的文件，以把在 sesame 中做过的改动合并进来：

```
aladdin> svn update
U   Number.txt
Updated to revision 3.
```

Subversion 挨着 Number.txt 打印了一个 U，让我们知道这个文件被更新过了。同时还告诉我们工作拷贝已经更新到了版本 3。如果我们再去看 Number.txt，就可以看到，已经额外地多出来了两行。

3.8 解决冲突

那么，要是两个人同时编辑同一个文件会发生什么？事实上，会有两种情况。第一种是两者的改动没有互相重叠。模拟这种情况要费些功夫，让我们来看看吧。

首先，编辑位于 sesame 目录中的那份 Number.txt。把第一行变成大写：

Number.txt(in Sesame)

```
ZERO
one
two
three
four
five
six
```

现在编辑位于 aladdin 中的那个版本的 Number.txt。这次把最后一行变成大写：

Number.txt(in Aladdin)

```
zero
one
two
three
four
five
SIX
```

我们刚才所做的就是模拟两个开发者，各自在本地修改了同一个文件。现在，这些改动是互相独立的，因为项目仓库对他们这两个中的哪个都不知情。让我们来改变这个情况。投硬币决定出 Aladdin 先签入他的改动：

```
aladdin> svn commit -m "Make 'six' important"
Sending        Number.txt
Transmitting file data .
Committed revision 4.
```

不久之后，sesame 的开发者也尝试着签入（记住，这个版本的文件是把第一行大写了的）。

```
sesame> svn commit -m "Zero needs emphasizing"
Sending        Number.txt
svn: Commit failed (details follow):
svn: Out of date: '/sesame/trunk/Number.txt' in transaction '7'
```

Subversion 告诉我们它尝试着提交 sesame 中的改动，但是失败了，因为/sesame/trunk/Number.txt 过期了。让我们尝试着从项目仓库更新本地的文件。记住我们的文件是大写的零，而项目仓库中的版本大写的是六。

```
sesame> svn update
G    Number.txt
Updated to revision 4.
```

Subversion 打印出 G 来告诉我们它已经把本地的版本与项目仓库的版本合并到了一块（之前，它打印的是 U 告诉我们它已经把本地拷贝更新到了项目仓库中的最新版本）。让我们看看本地的版本吧：

```
ZERO
one
two
three
four
five
SIX
```

神奇！我们的版本现在既包含了我们的改动，也包含了 Aladdin 的改动。我们同时编辑了同一个文件，是 Subversion 使得这一切成为了可能。

不要太洋洋得意了，我们本地的改动（大写的零）还没有存储到项目仓库中去呢。我们让 Subversion 提交我们的改动，这次它成功了，因为我们本地的版本包含了项目仓库最新的版本：

```
sesame> svn commit -m "Zero needs emphasizing"
Sending        Number.txt
Transmitting file data .
Committed revision 5.
```

下次 Aladdin 去更新的时候，他也会得到我们的改动的：

```
sesame> cd ..
work> cd aladdin
aladdin> svn update
U Number.txt
Updated to revision 5.
```

■ 令人头疼的事——改动产生了冲突

在前面的例子中，两个（虚拟）开发者做出的改动并没有互相重叠。要是两个开发者同时编辑同一个文件的同一行呢？让我们来试验一下。

到 sesame 目录中把 Number.txt 的第二行从 *one* 改成 *ichi*。暂时不要签入。现在再去 aladdin 目录把同一行从 *one* 改成 *uno*。让我们假定这次还是 Aladdin 先签入他的改动：

```
aladdin> svn commit -m "User likes Italian one"
Sending        Number.txt
Transmitting file data .
Committed revision 6.
```

现在回到 sesame 目录。记住假定的是两个彼此隔离开的用户，假定我们不知道 Aladdin 做过的改动，所以尝试着签入我们的改动：

```
sesame> svn commit -m "One should be Japanese"
Sending        Number.txt
svn: Commit failed (details follow):
svn: Out of date: '/sesame/trunk/Number.txt' in transaction 'c'
```

我们之前见过这样的消息：须要更新以获得项目仓库中新添加的改动：

```
sesame> svn update
C   Number.txt
Updated to revision 6.
```

Subversion 告诉我们把 Sesame 的工作拷贝更新到了版本 6 了，但是紧靠着 Number.txt 的 C 标志告诉我们当尝试着合并项目仓库的改动和本地的改动时有一个冲突存在。我们丢失了辛勤工作出来的劳动成果了么？没有。

给 CVS 用户的提示：当 CVS 检测到了一个冲突时，它会打印大量的警告消息，总的来说就是告诉你天塌下来了。它提醒你去修正这个冲突，因为在 CVS 中要签入一个仍然有冲突存在的文件是非常容易的。Subversion 跟踪了文件的状态，因此它知道你是否解决了冲突，而且在事情都一切正常之前它是不会让你签入的。

发生冲突往往是因为两个开发者有着某种误解。在本例中，一个开发者想要把那行改成意大利语的，而另外一个开发者则想要改成日语的。如果你考虑一下这个问题，就会发现这里的问题其实是沟通交流上的问题；在团队工作中存在着问题（或者至少是团队的开发流程中存在问题）。无论原因是什么，我们都得搞清楚“那行到底应该是什么？”Subversion 并不是先知的，

因此它无法解答这个问题。相反，它给文件添加了特殊的标记告诉我们是什么发生了冲突。在本例中，如果我们打开 Number.txt，会看到这样的内容：

```
ZERO
<<<<<<< .mine
ichi
=======
uno
>>>>>>> .r6
two
three
four
five
SIX
```

<<<<<<<和>>>>>>>标明了冲突发生的地方。在它们中间，可以看到我们的改动和与之相冲突的来自项目仓库的那个改动。

现在得做一些工作来搞清楚一下情况。要做的第一件事情是找出谁在项目仓库中做出了这个改动。我们使用 svn log 来帮助我们找出发生了什么。冲突标记似乎说的是 r6 导致了这个问题：

```
sesame> svn log -r6 Number.txt
------------------------------------------------------------------------
r6 | mike | 2004-09-08 23:01:03 -0600 (Wed, 08 Sep 2004)
User likes Italian one
------------------------------------------------------------------------
```

通过查看日志，我们可以看到改动的作者，以及他们签入时写的注释。我们走过去问他关于这个改动的事情。一个电话打到客户那很快就解决了这个问题：客户希望单词 *one* 是用日语写的，而 *two* 是用意大利语写的。Aladdin 一定是听错了。

了解了新情况之后，我们现在可以解决这个冲突了。编辑 sesame 目录中的 Number.txt，移除 Subversion 的冲突标记，并且按照客户想法把文件修改好：

```
ZERO
ichi
due
three
four
five
SIX
```

移除了冲突标记之后，可以告诉 Subversion 我们已经解决了冲突，然后提交这个文件：

```
sesame> svn resolved Number.txt
Resolved conflicted state of 'Number.txt'
```

```
sesame> svn commit -m "One is Japanese, two Italian"
Sending        Number.txt
Transmitting file data .
Committed revision 7.
```

事实上，Subversion 帮助我们发现了误解。我们解决了冲突，最后所有人都很高兴。乐观加锁取名乐观真是恰如其分啊。而且为了让事情显得不是那么令人害怕，须要强调一下冲突其实很少在真实项目中发生。

然而，也要注意 Subversion 没有读心术。可能发生这样的事情：两个人以两种不同的方式修正了同一个 bug。如果这些改动没有在源代码的层面发生冲突。Subversion 会乐滋滋地接受两者，虽然按道理来说在同一份代码中拥有两份代码修正同一个 bug 毫无道理。没有冲突意味着你与其他人没有在字面上发生碰撞，但是你仍然要依赖单元测试来验证改动是否有效正确。

Subversion 也为不可合并的文件，例如声音、图像和视频提供了严格加锁的支持。第 103 页的第 7 章“文件加锁和二进制文件”对文件加锁有更详细的讨论。

关于 Subversion 的快速浏览就到这儿。然而，你可能须要把你的测试项目仓库留好。后面，你可能会发现有些东西在真实的项目仓库中使用之前，先到测试项目仓库中试用一下是很有帮助的。

第 4 章

实例指导

How To...

虽然版本控制从理论上听起来不错，但是许多团队都没有使用它。有时这是因为理论似乎并没有很好地用实践解释清楚。从书本上读到“创造一个发布分支”是那么地轻而易举，但是在实践中你又需要输入什么样的 Subversion 命令才能正确地完成这个任务呢?

另外一个问题是，团队有时太过于注重版本控制，创造出了一个非常复杂的结构来存放源代码，编写了一个看起来很吓人的指令清单，而这一切却只是为了完成非常简单的任务。结果呢？最终（以我的经验来说还会非常快）团队会放弃；大家认为使用版本控制系统非常烦琐。

本书的其余章节就是讲这两个问题的。其中讲到了一种非常简单的组织版本控制系统的方式和每个团队都需要做的事情的最佳实践。我们建议你把最佳实践当作解决问题的灵丹妙药来用；无论何时，当你想要达到某种结果的时候，都需要遵循本书中的指导来做，而尽量不要偏离得太远；如果你发现你需要编造出一个我们没有包含的场景，那么在继续往下做之前请努力想想是不是有这个必要。有可能你并不需要它。

和任何的实例指导一样，你很快就会发现自己越来越习惯于照着它所说的那样来做，这时你就可以开始稍微做一些其他尝试了。然而，我们建议你

不要直接在真实项目的项目仓库中尝试什么新玩意。最好在测试项目仓库（例如我们在前一章中设置好的那个）中设置好场景，然后在那里做试验。

4.1 我们的基本哲学

我们认为版本控制是三大必要的技术实践之一；对于每个希望有效率的团队而言三者缺一不可（其他两个是务实的**单元测试**[HT03]、[HT04]以及务实的**项目自动化**[Cla04]）。每个团队都应该一直使用版本控制去管理所有产生的东西。因此，我们必须使它易用、直观，而且轻量（因为如果不是这样，人们最终会放弃不用它）。

简单性意味着简单的事情实际上就应该是简单的。签入我们的改动是一个简单（而且常见）的操作，因此，基本的操作应该只是一两个动作就能完成的。新创建一个给客户的发布版本是复杂一点的概念，因此多几步才能完成它没有什么关系，但是它仍然应该是尽可能简单的。

版本控制必须是直观的：我们需要把东西安排好，这样我们在做什么，用的是哪个版本的软件应该是能够一目了然的。

最后过程应该是轻量的；我们不希望版本控制成为完成真正要做的事情的阻碍。

4.2 使用版本控制系统的一些重要步骤

下面是用 Subversion 项目仓库来组织你的源代码的一些基本规则：

- 在你开始之前，要创建一个高效而且安全的访问项目仓库的方式。
- 你能在访问之后，学会每天都用得到的、简单的Subversion命令。
- 你公司开发的每个项目都必须存储在Subversion项目仓库的不同目

录中。你应该能够从一个地方签出项目的所有源代码。

- 如果项目包含互相独立开发的子模块，或者你希望在项目之间共享组件，那么这些组件应该作为单独的项目存放起来，并被作为外部资源包含到其他项目中去。

- 如果项目包含了第三方的东西(厂商不同,甚至还可能是开源项目),你需要把它们作为资源管理起来。

- 开发者应该使用分支把与主线开发具有不同生命周期的代码线从主线中分离出去，例如发布分支以及实验性质的代码分支。标签用来识别时间上的关键点，比如说一次发布或者修正了某个bug。

我们在接下来的章节中会逐个讲解这些问题。

第 5 章

访问项目仓库

Accessing a Repository

在第 29 页的第 3 章 “Subversion 入门” 中，我们创建了一个项目仓库并且学习了如何通过基于文件的 URL 来访问它。这对于个人用户来说是很好用的，但是并不适合整个团队一起协作开发。在本章中，我们要讨论的是三种主要的将项目仓库发布到网络上的方式，相应的访问方式，以及对应的优点和缺点。

第 157 页的附录 A 中有一个写给管理者的指南，讲述了安装、联网和配置 Subversion 安全策略方面的内容。

5.1 网络协议

创建好试验用的项目仓库之后，我们使用项目**仓库的** *URL* 来告诉 Subversion 想要签出什么。这个 URL 既包含了项目仓库的位置，也包括了我们感兴趣的路径。一旦工作拷贝创建好了，我们就不再需要使用项目仓库的 URL 了，因为 Subversion 记住了工作拷贝是从哪里来的。

repository URL

项目仓库的 URL 在任何想要直接访问的时候都用得上（比如说创建分支或者打标签的时候，或者合并大量的改动的时候）。图 5.1 展示了项目仓库的 URL 是如何组成的。

URL 的第一部分是 `file`。它指定了用来访问项目的 “**方式**”，在本

scheme

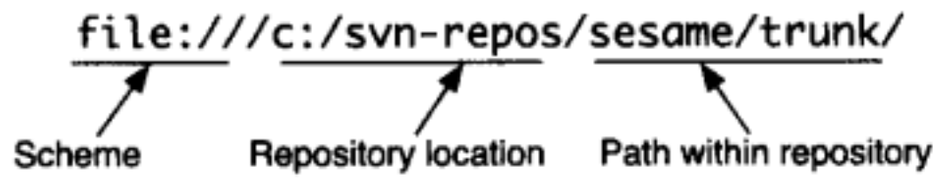


图 5.1 项目仓库 URL 的组成部分

例中是本地文件系统。下一部分，c:/svn-repos 告诉 Subversion 项目仓库的数据库文件在 C 盘的一个特定位置。最后，/sesame/trunk 指定了位于项目仓库内的。我们感兴趣的路径。

Subversion 在项目仓库 URL 中支持大量不同的访问方式，甚至还能让你自定义扩展。每个不同的访问方式告诉 Subversion 用不同的网络协议访问项目仓库。我们从最简单的 svn 协议开始。

■ svn

最简单的让一个项目仓库联网的方式就是使用 svn 访问。Subversion 自带了 svnserve，一个监听网络连接的小服务器，支持简单的用户认证。svnserve 可能是最适合那些想要快速上手，而且使用的网络又是私有的局域网的团队使用了。

如果管理员(可能就是你哦!)已经按照第 159 页的 A.2 节“使用 svnserve 联网”中的指导把 Sesame 项目仓库放在网上了，你可以通过运行下面的命令签出文件：

```
work> svn co svn://olio/sesame/trunk vizier
A  vizier/Number.txt
A  vizier/Day.txt
Checked out revision 7.
```

成功了！我们使用 svn 访问方式访问了一个称作 olio 机器上的项目仓库，而且把 Sesame 项目新签出到了一个叫作 vizier 的工作目录。

如果你在机器上工作目录中小试了一把，可能会发现 Subversion 不

让你提交任何改动。例如，尝试着添加一个叫做 Month.txt 的数据文件到项目中：

```
vizier> svn add Month.txt
A         Month.txt
vizier> svn commit -m "Added month data"
svn: Commit failed (details follow):
svn: Connection is read-only
```

这事如果发生了，那是因为你的管理员忘记给你项目仓库写权限了（默认是只读的）。让他们看看第 169 页的 A.5 节的“svnserve”，并且设置一些用户。一旦他们搞定了，你提交改动的时候应该会被要求输入用户名和密码：

```
vizier> svn commit -m "Added month data"
Authentication realm: <svn://olio:3690> sesame/trunk
Password for 'mike':
Adding          Month.txt
Transmitting file data .
Committed revision 8.
```

Subversion 决定用用户名 mike 试试，因为这是我机器上的用户名。如果这不正确，在密码提示符后面直接按回车，Subversion 会让你指定不同的用户名。

■ svn+ssh

用 svnserve 把项目仓库放在网络上的确很方便，但是它也有一些缺陷。首先，虽然密码并没有以明文在网络上传输，但是你的文件内容在传输的时候是没有加密的。任何嗅探你网络上流量的人都可以看到你文件的内容。这对处于同一局域网的团队来说可能没啥问题，但是如果你想要使用公共互联网来访问你的项目仓库就太不安全了。其次，密码是以明文方式储存在服务器的 conf 目录中的，而且只能由能够访问密码文件的管理员来修改。

Secure Shell

Subversion 通过使用 *Secure Shell*（SSH）来解决这两个问题。如果你是 Unix 用户，可能已经有了 SSH 的底层设施来连接你的服务器。SSH 使用了强大的加密机制来保护客户机服务器会话的内容。它被广泛用于管理互联网上的服务器。图 5.2 展示了 Subversion 如果使用 SSH 来保障 svn 连接的安全。

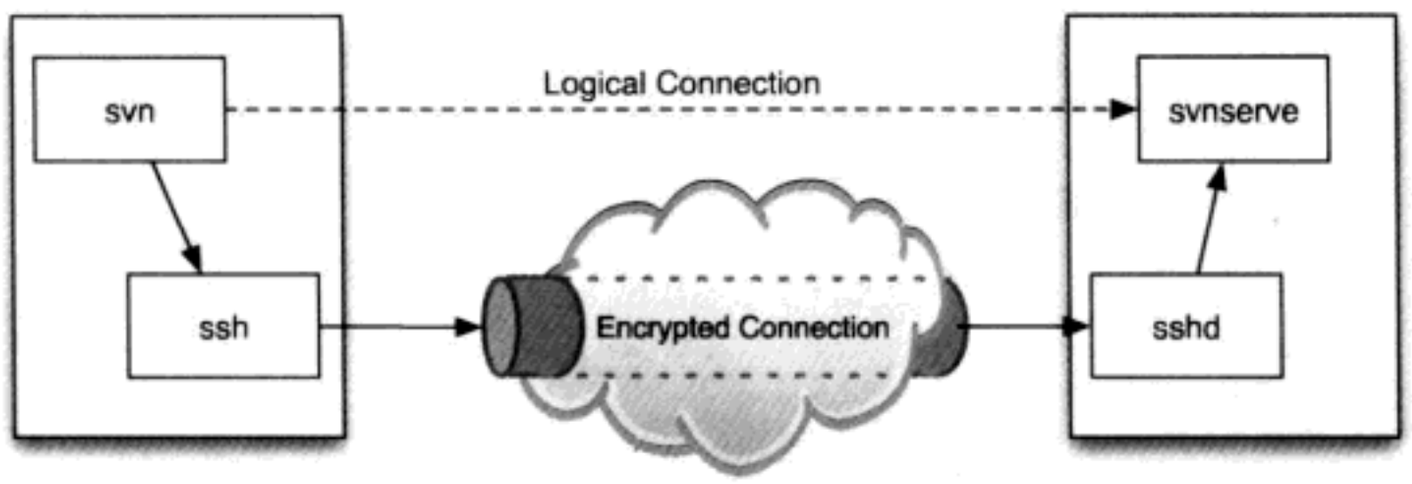


图 5.2 通过 SSH 把 Subversion 裹在安全通道中

Subversion 要求 SSH 客户端已经安装在你的机器上之后，才能让你使用 `svn+ssh` 访问项目仓库。Unix 的用户很可能已经安装过 SSH 了，但是如果你的工作在 Windows 上，就要做更多的工作了。Putty 是一个很好的 SSH 客户端，可以从 `http://www.chiark.greenend.org.uk/~sgtatham/putty/` 下载到。下载 `plink.exe`，把它保存在你的路径中；放在 `C:\Windows\system32` 下一般就没问题。如果你使用的是 TortoiseSVN，就无须担心 SSH 客户端的问题，因为 Tortoise 自带了 `TortoisePlink`。

接下来你需要编辑 Subversion 客户端的配置。Windows 程序把用户相关的数据储存在特殊的文件夹中，而其位置又取决于你的电脑是怎么安装的，使用的是什么版本的 Windows。如果你不能确定你的程序数据目录在哪里，请打开命令行输入下面的命令：

```
work> echo %APPDATA%
C:\Documents and Settings\mike\Application Data
```

一旦你找到了程序数据目录，打开 Subversion 子目录，编辑里面的 `config` 文件。编辑关于管道的那节，应该像这样：

```
[tunnels]
ssh=plink
```

如果你希望 Subversion 使用 SSH 来保护你的连接，请指定 `svn+ssh` 作为访问方式。如果你的服务器接受 SSH 连接，尝试运行：

```
work> svn checkout \
          svn+ssh://olio/home/mike/svn-repos/sesame/trunk \
          princess
mike@olio's password:
A  princess/Month.txt
A  princess/Number.txt
A  princess/Day.txt
Checked out revision 8.
```

这看起来和之前用 svnserve 时用到的 URL 差不多，只是把访问方式变成了 svn+ssh。如果你访问项目仓库时遇到了问题，请参见第 162 页 A.3 节的“SSH 常见问题”，那里有一个诊断相关问题的指南。

Subversion 现在使用了 SSH 来打开与服务器的连接，并且把你作为 Unix 用户来验证身份。Subversion 使用标准的 Unix 用户和组权限来决定连接的用户是否有权限访问项目仓库。如果你使用 SSH 公有/私有密钥或者 SSH 代理来管理你的证书，Subversion 客户端自动就会利用这些，也就是说，你不会被提示要求输入密码了。

使用 svn+ssh 是很有诱惑力的选择，如果你的用户已经有了 SSH 账号，就可以利用你现有的基础设施。额外的安全机制让你能无忧地连接到互联网，不用担心有人会窃取你 Sesame 项目的代码，也不会有要设置一个 VPN 的烦恼。svn+ssh 是很自然的解决方案，而且运行起来也很快。

■ http

Subversion 还能通过使用 Apache web 服务器把项目仓库架设在 web 上。一个名叫 mod_dav_svn 的 Subversion 模块帮我们把所有的工作都做好了，让 Subversion 能够与传统的 web 站点共享 web 服务器。Apache 是高度可配置的，而 Subversion 利用了其内建的安全机制和可缩放性。你可以使用标准的 http 和 https 架设项目仓库，利用 Apache 支持的所有验证机制。

你可能听说过 Subversion “需要” Apache，这实际上是不正确的；svn，svn+ssh 都不需要任何额外的东西就能让你的项目仓库联网。大部分预先打

好的 Unix 包都依赖于 Apache，是因为它们安装了所有三种联网选项，这也就是为什么会有这种误解的原因。连同 Apache 一起使用 Subversion 可能是在互联网上共享项目仓库最流行的做法了。

Apache 给用户提供了大量验证身份的选项。从使用密码文件这种最基本的验证方式到与 Windows 域或者 LDAP 服务器集成的高级验证方式，Apache 是超级灵活的。你甚至可以设置基于目录的安全，把你的项目仓库划分出只读甚至完全私有的区域。你可以利用标准 SSL 证书来加密服务器连接，同时使用标准的 web 服务器端口号以避免防火墙带来的麻烦。

为了访问一个架设在名为 `olio` 的 Apache 服务器，可使用以下命令：

```
work> svn checkout \
          http://olio.mynetwork.net/svn-repos/sesame/trunk \
          sesame
Authentication realm: ... Subversion repository
Password for 'mike': ******
A  sesame/Month.txt
A  sesame/Number.txt
A  sesame/Day.txt
Checked out revision 8.
```

这个项目仓库甚至对于已经验证身份的用户也是只读的。Subversion 自动会去尝试用户名 `mike`；如果不对，只要直接按回车，不用输入密码，这样 Subversion 就会让你指定用户名了。

5.2 选择如何联网

Subversion 的三种联网协议（`svn`，`svn+ssh` 以及 `http`）对于安装、安全和管理三个方面有着不同权衡。你选择哪种取决于你已经有了什么样的底层设置，取决于你的安全性要求以及你是否熟悉 Apache。

值得注意的一点是你今天选择的联网选择并不一定是你明天一定要用的。把项目仓库联网不过是把它放在网上而已，比如你可以在 `svnserve` 和 Apache 之间频繁地自由切换。而且“**同时**”支持多种不同的访问机制也是可

能的，虽然你必须对权限问题格外小心。

如果你的团队是在一个相对安全的局域网内，或者是更大一点的用 VPN 互相连接的网络中，使用简单的 `svnserve` 服务器以及 svn 协议可以快速地设置和运行起 Subversion。你在添加新用户或者修改已有用户密码时会有一些额外负担，但是相比起快速地上手来说是值得的。Subversion 1.3 添加了基于目录的授权方式使得对于在同一局域网内的团队来说 `svnserve` 几乎和 Apache 一样灵活。

如果你有现成的 SSH 基础设施，使用 `svn+ssh` 是不错的选择。你的连接受到了强大的加密保护并且可以利用 SSH 提供的关键的密钥管理和验证选项上的方便。虽然在使用这种方式之前，得先确保你的 Unix 管理员了解组、umask、以及 sticky bit 的含义。

如果你想要在互联网上架设项目仓库，利用 Apache 的丰富的验证机制，或者只是拿这个大家伙玩玩，跑一个“真实”的服务器，使用 Apache 来架设 Subversion 的项目仓库正是你的正确之选。你能够使用 SSL 和客户端服务器证书来加密和验证你确实是在和你需要通话的人通话，而且你可以使用 Windows 域、LDAP 或者任何 Apache 支持的验证机制来给用户授权。你还能通过 Apache 的 `mod_authz_svn` 模块更精确地指定项目仓库的哪些部分用户可以访问。在标准 HTTP 端口使用 Apache 还意味着你在防火墙上开的洞可以更少。你的网管会为此感谢你的。

PDG

第 6 章

常见的 Subversion 命令

Common Subversion Commands

在第 29 页的第 3 章“Subversion 入门”中，我们创建了一个简单的项目并且试验了一些基本的 Subversion 命令。在本章中，我们将通过一系列的实例指导来安排内容：这些 Subversion 命令是你每天都会用到的常见命令。

本节并不复杂。本书的后面会有一些更高级的内容，例如对发布的管理工作目录以及第三方代码等。然而，本章中的命令和技术应该可以让你完成使用 Subversion 时所要做的 90%的工作。

这些例子假设你已经设置好了你的项目仓库并且能够访问网络。我们假设 Sesame 项目的主线代码（主干）位于 `svn://olio/sesame/trunk`。你需要用自己的服务器名替换 `olio`，而且如果你使用的是 `http` 或者 `svn+ssh`，则要把 `svn` 替换为正确的访问方式。

6.1 把东西签出来

`svn checkout` 命令（经常简写为 `co`）让 Subversion 能从项目仓库的一个目录中取出东西并新创建一个工作拷贝。如果使用的是最简单的格式，签出命令会把工作拷贝创建在一个与项目仓库目录同名的目录中：

```
work> svn checkout svn://olio/sesame/trunk
A  trunk/Month.txt
A  trunk/Number.txt
A  trunk/Day.txt
Checked out revision 8.
```

现在，Subversion 在本地的 trunk 目录创建了工作拷贝，之所以叫 trunk，是因为那是项目仓库中的目录名。这可能不是你想要的，特别是你想要遵循第 111 页第 8 章“组织你的项目仓库”中推荐的命名规范的话。在签出的时候，你可以使用额外的参数来指定目录名，Subversion 会使用这个目录名来创建工作拷贝：

```
work> svn checkout svn://olio/sesame/trunk sesame
A  sesame/Month.txt
A  sesame/Number.txt
A  sesame/Day.txt
Checked out revision 8.
```

默认情况下，Subversion 签出的是项目仓库中最新的版本。如果你想要一个过去的版本，使用 -r 选项去指定版本号或者你想要的日期。第 83 页的 6.6 节“使用 Subversion 版本标识符”对于如何引用一个特定的版本有更详细的描述：

要签出一个我们添加 Month.txt 之前的 Sesame 项目拷贝，指定版本 7：

```
work> svn checkout -r 7 svn://olio/sesame/trunk old-sesame
A  old-sesame/Number.txt
A  old-sesame/Day.txt
Checked out revision 7.
```

如果你像我们习惯的一样，可能最终在工作目录中会有大量不同的工作拷贝。要搞清楚工作拷贝是从哪里来的，可使用 svn info 命令：

```
work> svn info sesame
Path: sesame
URL: svn://olio/sesame/trunk
Repository UUID: d6959e13-a0e3-0310-8d55-a8c2e0b5e323
Revision: 34
Node Kind: directory
Schedule: normal
Last Changed Author: mike
Last Changed Rev: 7
Last Changed Date: 2004-10-05 13:07:15 -0700 (Tue 5 Oct 2004)
```

这里的重点是第二行的 URL。Subversion 告诉你本地的 sesame 目录其实原始来源是 svn://olio/sesame/trunk/。

6.2 保持更新

如果你不是唯一一个在项目上工作的人，在你工作的时候就很有可能其他人已经更新过项目仓库了。频繁地更新你的工作拷贝，不断合并他人的改动是一个不错的主意；你把它留在那儿越久，修正冲突的麻烦就越大。[1]我们一般一个小时左右更新一次工作拷贝。

在工作拷贝中使用 `svn update`，并从项目仓库里更新目录中的所有文件（以及它的子目录）。添加到项目仓库中的文件和目录会被添加到工作拷贝中，从项目仓库中移除的文件和目录也会从工作拷贝中移除。下面的命令更新了 Sesame 项目的工作拷贝：

```
work> cd sesame
sesame> svn update
```

你可以选择只更新你签出的目录树的一部分。如果你在项目的子目录中执行命令，那么只有在这个位置之下的文件才会被更新。这可能会节省时间，但也有一定的危险，因为你可能没有得到所有的文件。

你还能指定一个或者多个文件或者目录，通过在命令行上指定它们的名字来更新它们：

```
main> svn update build.xml src/ test/
```

在更新的过程中，Subversion 会显示每个文件的状态以及对应的重要活动。例如，下面是在更新一个包含了 Subversion 自身的源代码的目录时产生的日志：

```
subversion> svn update
U    include/svn_repos.h
G    libsvn_client/status.c
A    bindings/java/javahl/build
A    bindings/java/javahl/build/build.xml
U    bindings/swig/perl/native/Repos.pm
```

---

[1] 频繁地合并还有另外一个作用。如果另外一个开发者走到岔路上去了，或者他们的改动从长期的眼光来看一定会出问题，如果你经常合并，你很快就会发现这些问题。你越早得到这些反馈，修正这些问题的痛苦就越小。

```
U   bindings/swig/perl/native/Base.pm
A   bindings/swig/perl/native/Makefile.PL.in
UU  bindings/swig/perl/native/h2i.pl
U   bindings/swig/perl/native/Ra.pm
D   bindings/swig/perl/native/Makefile.PL
 U  bindings/swig/perl/native
U   clients/cmdline/propedit-cmd.c
A   po/pt_BR.po
U   po/zh_TW.po
Updated to revision 11141.
```

Subversion 打印出以下的字符来表明对每个文件和目录都做了什么：

- A代表项目仓库中有新文件时，Subversion为了更新你的工作拷贝，添加了一个文件到其中。
- U表示你工作拷贝中的文件过期了，因为有一个更新的版本签入到项目仓库中。Subversion已经把你工作拷贝的文件更新为新的版本。
- D代表因为该文件已经从项目仓库中删除了，Subversion把它从工作拷贝移除了。
- G表示在你工作拷贝中的文件已经过期了，而且你本地还做了修改。Subversion成功地把项目仓库中的版本和你本地的修改合并到了一起。
- C表示你工作拷贝中的文件过期了，而且你本地也做了修改。Subversion尝试着合并项目仓库的改动和你本地的修改，但是遇到了冲突。你需要把这个冲突解决了之后才能签入。

你可能注意到了 Subversion 打印在 `h2i.pl` 前面的 `UU` 以及 `bindings/swig/perl/native` 目录前的空格和 `U`。这不是打错了，Subversion 其实是在打印两行信息。第二行代表文件的属性，而不是文件本身。Subversion 属性在 6.4 节“属性”中有更具体的讨论。

6.3 添加文件和目录

svn add 命令告诉 Subversion 添加文件和目录到项目仓库之中。当你添加目录的时候，Subversion 自动添加了目录中的所有文件，以及其子目录，除非你指定--non-recursive 选项：

```
sesame> mkdir timelib
sesame> cd timelib
timelib> # ..create and edit Time.java..#
timelib> cd ..
sesame> svn add timelib
A           timelib
A           timelib/Time.java
```

注意此时 Subversion 只记住了你想要添加到项目仓库中的文件的名字；它并没有实际添加文件或者把改动让团队的其他人也能看见。你须要使用 svn commit 命令去把新文件提交到项目仓库中。

Subversion 在项目仓库中以二进制格式储存所有文件，使用高效的比较二进制差异的算法来计算版本之间的改动。这对于文本和真正的二进制数据都很有效，因此，你无须在添加文件到项目仓库时告诉 Subversion 文件是不是二进制的。

在第 76 页的 6.4 节“设置 Mime 类型”中，我们会看到 Subversion 区别对待文本和二进制文件。这也就意味着有时值得检查一下 Subversion 是不是检查出文件是二进制了。当你添加一个文件到 Subversion 时，如果它是二进制的文件，Subversion 会自动把属性 svn:mime-type 设置为 application/octet-stream。下一节会具体讲述关于“属性”的内容。

6.4 属性

当我们把大部分注意力都放在 Subversion 如何储存文件的“**内容**”的同时，也不要忘记了它还能把与文件（以及目录）关联的元数据存储到项目仓库中。[2] Subversion 把这些元数据称作“**属性**”，而且管理属性改动的方式和 *properties*

[2] Subversion 实际上还能储存版本的属性。例如，每次提交的与文件关联的元数据是存储在版本的文本属性之中的。

管理文件内容的方式是一样的。属性可以被不同用户修改，而且在每个工作目录中当用户运行 `svn update` 的时候会被更新。这会导致合并或者冲突，就像改变文件内容时可能遇到的那样。

属性用简单的字符串命名，而且可以包含普通文件可以包含的任何内容，特别要指出的是这也包括二进制内容。属性可以用来给文件关联额外的数据，以及任何你想要的格式。例如，Java 的源代码可以有一个关联的“审阅者”的属性，告诉你谁最后审阅过这个文件。一个储存音乐文件的项目仓库可能在二进制属性中储存每个文件的一小段样本，而不是把样本文件储存在主文件的旁边然后用命名习惯来关联两者。

只要你想，任何时候你都可以使用 Subversion 属性，但是你应该知道一些特殊的属性。这些属性可以改变 Subversion 处理文件的行为，而且都以 `svn:` 前缀开头。

■ 操作属性

要设置一个文件的属性，使用 `svn propset` 命令：

```
sesame> svn propset checked-by "Mike Mason" Number.txt
property 'checked-by' set on 'Number.txt'
sesame> svn status
 M      Number.txt
```

这儿，我们把 `Number.txt` 的 `checked-by` 属性的值设置为 `Mike Mason`。可能我们项目的发布过程需要每个文件都设置这个属性，从而我们可以知道谁批准了对应的内容。值得注意的一点是我们改动的是文件的属性，而 Subversion 对它的处理与对内容改动的处理是一样的。Subversion 记录我们文件的本地拷贝状态为已修改，而且如果我们想要其他人都能看见它的话，需要把改动提交到项目仓库。

要编辑一个属性，请使用 `svn propedit`。这会打开一个编辑器从而你可以轻松地管理多行的文本属性：

```
sesame> svn propedit checked-by Number.txt
        # .. edit the property, then save and quit the editor ..
Set new value for property 'checked-by' on 'Number.txt'
```

svn proplist 和 svn propget 可以列出当前文件的所有属性并打印出属性的当前值：

```
sesame> svn proplist Number.txt
Properties on 'Number.txt':
  checked-by
sesame> svn propget checked-by Number.txt
Mike
Ian
```

最后，你还可以使用 svn propdel 来彻底删除一个属性。记住属性并没有永远丢失，Subversion 跟踪属性的改动就像跟踪文件的内容一样，因此你总是可以回溯到以前，找出之前的任意版本。

■ 关键字展开

keyword expansion

如果你曾经用过别的版本控制系统，可能熟悉“**关键字展开**”这个概念。这大致上是说让你的版本控制系统在它签出文件并更新文件时修改你的工作拷贝，从而它能为你填充有用的信息。每个这样有用的信息片段被一个“**关键字**”代表，通常是括在货币符号之中，它们统一地放置在每个储存在版本控制系统的文件中。Subversion 中的关键字是以未展开形式存放在项目仓库中的，这样做的目的是为了更容易查找差异和合并文件。

keyword

我非常“不”推荐你使用 Subversion 的这个特性，因为它会给你带来很多烦恼。我曾经想把书的这页变成活页，这样读者可以把它扯掉，彻底地忘记关于关键字展开的一切，但是我们的印刷商说那样做生产成本太高了……

要启用关键字展开，你需要给每个包含关键字的文件设置 svn:keywords 属性。其属性值应该列出你想要给该文件展开的所有关键字。Subversion 提供了以下关键字：

$LastChangedDate$

也被简写为$Date$，这个关键字描述了文件最后一次提交到项目仓库的时间。它展开为类似 2004-09-26 18:11:03 -0700(Sun, 26 Sep 2004)这样的字符串。

$LastChangedRevision$

也被称作$Revision$或者Rev，这个关键字展开为文件最后一次提交到项目仓库的版本号。

`$LastChangedBy$`

也被缩写为`$Author$`，这个关键字展开为最后一个提交文件的用户名字。

`$HeadURL$`

也被缩写为`$URL$`，这个关键字展开为文件在项目仓库中的完整URL。

`$Id$`

这个关键字展开为其他关键字的简短摘要，适合用在文件的头部。

假设我们想要给 Sesame 项目中的 `Number.txt` 文件启用关键字展开。首先，需要设置 `svn:keywords` 为想要展开的关键字列表：

```
sesame> svn propset svn:keywords "HeadURL Id" Number.txt
property 'svn:keywords' set on 'Number.txt'
```

现在编辑 `Number.txt` 文件，在头部添加上两行要展开的关键字。这儿使用的是`$HeadURL$`和`$Id$`：

```
# $HeadURL$
# $Id$
ZERO
ichi
due
three
four
five
SIX
```

当提交我们的改动时，Subversion 会注意到要求了关键字展开并且修改了工作拷贝的文件。每个关键字被展开为 Subversion 对那个文件持有的最新信息：

```
sesame> svn commit -m "Added file keywords"
Sending        Number.txt
Transmitting file data .
Committed revision 10.
sesame> cat Number.txt
# $HeadURL: svn://olio/sesame/trunk/Number.txt $
# $Id: Number.txt 10 2004-09-27 00:09:05Z mike $
ZERO
ichi
due
three
four
five
SIX
```

\// Joe 问……
Log关键字在哪儿

包括 CVS 在内的其他版本控制系统都有关键字一说，经常包含一个Log关键字。它展开为一个在提交这个文件改动时使用过的日志消息的列表。

实践中遇到的问题是源代码中所有的这些外加的东西阻碍了阅读源代码。我曾经看见过源代码有两页甚至三页的日志消息在它的顶上，你要看完这些才能读到真正代码。代码放在那儿是被阅读的，任何阻碍了我们阅读的都是不好的东西。

问题是信息重复。任何可以使用关键字插入的东西都已经储存在 Subversion 之中了（它不得不这样；否则 Subversion 也就没法添加它们了）。因此为什么不直接去 Subversion 那里获得这些信息呢？那样的话，你可以得到权威的信息，而且保证是最新的。

Subversion 的开发者认为关键字，特别是可能很长的提交日志，使用不应该提倡。结果就是他们没有包含Log关键字。

关键字展开真的没有什么好处，而且它有几项缺点。我们推荐不使用它。

在使用 Pragmatic Programmers 的基于 CVS 的系统撰写本书时，Mike 被Log和Id关键字烦死了，不得不在编写很多章时把它关掉。使用 CVS 来编写 Subversion 书。

编辑的注释：Mike 本书的第一版深深地打动了我们，所以我们现在使用 Subversion 来编写所有的图书了。

autoprops

如果你想要在很多文件中使用关键字展开，比如说你所有的 `.java` 文件，那么有很多属性须记得设置。幸运的是，Subversion 有一个称作 *autoprops* 的功能可以帮你设置这些属性。Autoprops 在第 77 页的 “自动设置属性” 中有具体的讲述。

■ 忽略某些文件

大部分时间，你的工作拷贝中既包含你想要的版本控制文件（源代码、构建脚本、你的程序图片等等），也包含你觉得放在那里挺好而且不需要储存在项目仓库中的文件（临时文件、编译了的代码以及日志文件）。一些 Subversion 命令，比如 `svn status`、`svn add` 和 `svn import`，假设你关注你工作拷贝中的所有文件。例如，当你忘记添加一些文件时，`svn status` 会把它们列出来。

如果它们真是你不想要 Subversion 管理的，总是看到为这些文件打印出来的额外信息感到很烦人，而且还可能是危险的（你试试看不小心添加一堆大的临时文件到你的项目仓库中，你的管理员会不会跑过来杀了你……）。幸运的是，有一个简单的办法可以避免这个问题：设置目录的 `svn:ignore` 属性，指定你想要 Subversion 忽略的文件。

假设我们在开发 Sesame 项目中的一个有关时间的库。让 Subversion 报告当前状态，我们可以看到：

```
sesame> svn status timelib/
?      timelib/Time.class
?      timelib/Time.java.bak
M      timelib/Time.java
```

这儿，可以看到改动了 `Time.java`，但是 Subversion 还提到了 `Time.class` 和 `Time.java.bak`，这些都不是我们所关心的。

使用 `svn propedit svn:ignore timelib` 来打开一个编辑器编辑 `timelib` 的 `svn:ignore` 属性。输入以下内容：

```
*.class
*.bak
```

现在运行 `svn status`，`.class` 和 `.bak` 文件就会被忽略掉了：

```
sesame> svn status
 M     timelib
M      timelib/Time.java
```

timelib 目录被列为被修改过了，因为我们修改了它的 svn:ignore 属性。

一旦你的改动被提交了，每个人都会接收到 timelib 更新过的 svn:ignore 属性，从而他们的 Subversion 也会忽略掉工作拷贝中的那些文件。svn:ignore 属性只应用于指定目录中的内容；它不会递归地应用于其子目录。

■ 设置换行方式

control characters

计算机系统使用普通字符（如字母、数字等）以及特殊的“**控制字符**”组合来存储文本文件。这些字符中有一种是用来标记文本的行结尾的，其形式是一个字符或两个字符的组合。取决于是什么操作系统，计算机会使用 carriage-return 后跟一个 linefeed（CRLF，用于 Windows 计算机），或者只是 linefeed（LF，用于 Unix 和 Mac OS X），或者有时只是 carriage-return（CR，用于老版本的 Mac OS）。

如果你储存的文件要在处理换行方式不同的客户机器上使用，你就要注意行结尾是如何被存储的。Subversion 储存的所有文件，无论它们是文本、图片、编译过的代码，或者是电影，在项目仓库中使用的都是二进制格式。除非你要求它这么做，Subversion 绝不会转换文件的换行方式，也就是说，你可以完全地忽略本节。

如果你确实需要在不同操作系统之间共享文件，你可能已经看到了奇怪的结果。比如说，使用 Windows 的记事本打开 Unix 格式的文件，产生的文件是许多小方块，而不是新行。在 Unix 上打开 Windows 格式的文件可能导致在每行结尾有一个^M 字符。

你的编辑器或者 IDE 可能说自己能够为你做这样的转换，或者在编辑它的时候不改变其现有的换行方式。我们时常还会发现最好的事情是在不同的操作系统上保持其本地的换行方式。如果你设置了 svn:eol-style 属性为以下表格中的一个值，Subversion 会给你做这样的转换。

本地	Subversion 会把换行字符翻译为客户机操作系统希望的风格，也就是说会在 Windows 上使用 CRLF，在 Unix 上使用 LF。
CRLF	在工作拷贝中创建文件时，Subversion 会始终使用 CRLF 作为换行字符。
LF	Subversion 会始终使用 LF 作为换行字符。
CR	Subversion 会始终使用 CR 作为换行字符。

■ 设置 Mime 类型

设置文件的 `svn:mime-type` 属性告诉 Subversion 这个文件的确切类型。Mime 类型在互联网上被大量使用，特别是电子邮件和 web 服务器，用于描述被传输的文件或者用于描述所发邮件的附件。例如，XML 文档的 mime 类型是 *text/xml*，JPEG 图像的类型是 *image/jpeg*，而微软 Word 文档的 mime 类型是 *application/msword*。

设置文件的 `svn:mime-type` 有以下好处：

首先，Subversion 假设文件的 mime 类型不是文本（以 *text/*开头），其内容就是二进制数据，因此它在合并文件和显示版本差别时，会区别对待。二进制文件的版本差异对人来说是不可读的，所以 Subversion 会跳过显示差异这步直接告诉你文件被改动了。而且合并两个二进制文件也可能不好使，所以，当你从项目仓库接收改动到你工作目录中改动过的二进制文件时，Subversion 把你改动过的版本文件改名为 `.orig` 后缀，并且用从项目仓库接收到的新数据替换你的文件。

其次，当 Subversion 把 Apache 用作其网络服务器的时候，你可以使用普通的 web 浏览器浏览项目仓库。当你点击一个链接的时候，Apache 把文件返回给浏览器，Subversion 使用 `svn:mime-type` 属性来识别文件的类型。

设置该属性有助于避免你在点击项目仓库中的压缩文件时看到满屏的二进制数据。

■ 可执行标志

有些操作系统，尤其是 Unix，把普通的数据文件和程序文件区别对待。为了能够在 Unix 上运行一个程序，必须设置一个“可执行标志位”。如果你签入到项目仓库中的是可执行文件或者脚本，用户签出这些文件的时候是不会自动设置其可执行标志位的。设置文件的 `svn:executable` 属性就可以让 Subversion 在任何时候文件被签出时，都会自动设置其可执行标志位。这个属性设置什么值都没有关系，只要你设置了它，Subversion 就会为你设置可执行标志位。

在 Windows 上，所有的文件都是可执行的，所以你可能就不需要关心这个了。

■ 自动设置属性

Subversion 属性设置是非常有用的，但是困难的是我们需要一个个设置需要的属性给每个关心的文件或者目录。很容易会忘记设置其中某个属性，而且这可能会在以后导致问题。

幸运的是，Subversion 有一个叫做 *autoprops* 的特性，它可以让你指定自动添加的属性。例如，你可能决定把所有的 `.java` 文件的 `svn:keywords` 属性都设置为 *LastChangedDate*，以及 `svn:eol-style` 都设置为 *native*。你还可能决定无论何时添加一个 `.gpg` 文件都设置其 `svn:mime-type` 为 *application/pgp-encrypted*。

自动属性是客户端的设置，因此，如果想要你的所有开发者都使用它们，就须制定某种规章制度以确保每个人都能正确设置。不幸的是，Subversion 还没有能力把配置设置从服务器“广播”到客户端，因此你必须手工来做。

Subversion 把你的设置储存在一个与用户相关的程序数据目录中。这个目录具体在哪里取决于你使用的是 Unix 还是 Windows。第 59 页的 5.1 节

"svn+ssh"讲过在 Windows 下如何查找这个目录，而在 Unix 下，Subversion 使用的是~/.subversion。

编辑 Subversion 的 config 目录，去掉下面这行注释：

```
enable-auto-props = yes
```

往下再滚一点，去掉 autoprops 节的注释，把你想要的属性添加在这儿。为了启用.gpg 文件的 mime 类型，以及设置.java 的换行方式为本地，你需要把这节改成这样：

```
[auto-props]
*.java = svn:eol-style=native
*.gpg = svn:mime-type=application/pgp-encrypted
```

6.5 拷贝、移动文件和目录

Subversion 记住每个你曾经提交到项目仓库中的文件和目录。在大部分情况下，这都是很好的事情，但是如果你犯了个错误，把文件添加到了错误的目录中，或者添加错了名字。现代的编程方式还包括一种称为**重构**的技术，使用这种技术，你经常要在想到了更好的名字来描述这个文件干了什么的时候来重命名项目文件，或者给它在项目中找到更符合逻辑的位置。

幸运的是，Subversion 中有拷贝和移动命令让你来移动和重命名[3]文件和目录。Subversion 跟踪历史的功能也知道这些操作的存在，所以要移动文件的话，最好使用 Subversion 的命令而不要自己手工去移。

■ 拷贝文件

你可以使用 Windows 浏览器手工拷贝文件，或者使用 Unix 的 cp 命令，然后把新文件添加到版本控制中， Subversion 提供了 svn copy 命令让你做同样的事情。

---

[3] "重命名"不过是"移动"，只不过正好是把文件移动到了同一个目录而已。Unix 大师可以解释为什么这是有道理的，至少 Subversion 的 move 和 rename 做的其实是一样的事情。

拷贝在 Subversion 中是基础性的操作，很多东西都是基于这个命令的。文件的后续版本是文件的拷贝，然后把内容改变。分支是把整个目录拷贝到一个新的位置。标签为一些文件的拷贝提供了项目仓库在某个特定时刻的快照。

拷贝是如此方便，那你为什么还要在你的项目仓库中拷贝出一份新文件来呢？极端而言，你可能不需要。但你要是使用 `svn copy` 拷贝了一个文件的话，对于原本和拷贝的历史，Subversion 会跟踪到同一个来源。事实上，Subversion 甚至没有储存文件的完整拷贝；它只是存储了对它拷贝来的文件的引用。当你有许多文件从同一个来源拷贝而来，却只是改动了很少一点东西的时候就会觉得这很有用了。

广告已经做得够多了。使用 `svn copy` 是不错的想法，而且它是这么用的：

```
sesame> svn copy Number.txt Data.txt
A         Data.txt
sesame> svn commit -m "Created example data file"
Adding          Data.txt
Committed revision 24.
```

在你的本地工作目录中拷贝一个文件或者目录会创建拷贝，并且把它们列到要添加到项目仓库中的文件列表中去。一个 `svn commit` 就可以签入它们并完成这次拷贝了。

因为 Subversion 记住了文件的共享历史，询问 `Data.txt` 的历史也会给出 `Number.txt` 的历史：

```
sesame> svn log Data.txt
------------------------------------------------------------------------
r24 | mike | 2004-11-17 16:00:37 -0700 (Wed, 17 Nov 2004)
Created example data file
------------------------------------------------------------------------
r11 | mike | 2004-10-04 21:05:37 -0600 (Mon, 04 Oct 2004)
Added Ian as reviewer
------------------------------------------------------------------------
r10 | mike | 2004-09-26 18:09:05 -0600 (Sun, 26 Sep 2004)
Added file keywords
------------------------------------------------------------------------
r7 | mike | 2004-09-08 23:22:06 -0600 (Wed, 08 Sep 2004)
One is Japanese, two Italian
------------------------------------------------------------------------
```

■ 重命名文件

假设 Sesame 项目的 `Time.java` 的行为实际上已经更类似“时钟”，所以我们想要给它改个名字。我们可以使用 Windows 浏览器或者 Unix 的 `mv` 命令在我们的工作拷贝中重命名文件，然后再使用 `svn delete` 删除 `Time.java` 以及 `svn add` 添加 `Clock.java`，但是这就没法让 Subversion 为我们跟踪这个文件的历史了。

首先，让我们看看 `Time.java` 的历史：

```
timelib> svn log Time.java
------------------------------------------------------------------------
r14 | mike | 2004-10-04 21:12:48 -0600 (Mon, 04 Oct 2004)
Added freeze/unfreeze time methods
------------------------------------------------------------------------
r13 | mike | 2004-10-04 21:10:50 -0600 (Mon, 04 Oct 2004)
Added getCurrentDate() method
------------------------------------------------------------------------
```

最近我们对这个文件的改动是添加了冻结和解冻系统时间的方法。实际上，这个类最好的名字应该是 `Clock`。我们知道，如果类的名字取得不好，事情可能会越来越不好控制，所以我们决定及早地做出改动而不是拖到以后再做。使用 `svn move` 命令来重命名文件：

```
timelib> svn move Time.java Clock.java
A         Clock.java
D         Time.java
```

这儿，Subversion 告诉我们“移动”实际上是添加和删除。有朝一日，Subversion 可能会把重命名作为一等公民来支持，但是目前 Subversion 的移动在项目仓库中是以从旧名字拷贝一份到新名字，然后再删除旧名字的形式实现的。

你先别兴奋过度，直接就提交改动，你应该先运行一遍单元测试确保没有破坏任何东西。至少，这个 Java 文件现在无法编译，因为它在称为 `Clock.java` 的文件中包含了一个 `Time` 类。打开你最喜欢的编辑器，把类的名字改为 `Clock`。然后再确认一次你的测试还能通过。你可能须要改动引用了这个类的代码，让它们也使用这个新名字。[4]

[4] 在 Java 文件中重命名需要好几个步骤，就像用其他编程语言一样。幸运的是，一些开发环境直接集成了版本控制，它们可以为你自动地执行所有的重命名，添加和删除操作。去看看你的 IDE 是不是支持重构吧。

什么是名字

Subversion 的 move 命令也可以被称作 svn mv、rename 以及 ren。svn copy 命令可以简写为 svn cp，如果你喜欢简短的话。

当我们在谈论命名这个话题的时候，值得指出的一点是给东西（如类、变量、方法、测试、数据文件、机器、进程等）命名是相当困难同时又很重要的事情。大部分人都不能在第一次就完全正确地给东西取好名字，但是一个名字取得恰当，有助于避免误解和加快交流。一旦你意识到对某件东西有更好的名字，就立刻花时间去重命名它。你的同事会感谢你的！

一旦所有东西都一切正常了，提交你的改动：

```
timelib> svn commit -m "Renamed Time to Clock"
Adding           timelib/Clock.java
Deleting         timelib/Time.java
Transmitting file data .
Committed revision 15.
```

现在如果你查看新的 Clock.java，我们可以看到努力得到了回报，因为 Subversion 在重命名之后保留了之前的历史：

```
timelib> svn log -v Clock.java
------------------------------------------------------------------------
r15 | mike | 2004-10-04 21:13:40 -0600 (Mon, 04 Oct 2004)
Changed paths:
    A /sesame/trunk/timelib/Clock.java
                (from /sesame/trunk/timelib/Time.java:14)
    D /sesame/trunk/timelib/Time.java
Renamed Time to Clock
------------------------------------------------------------------------
r14 | mike | 2004-10-04 21:12:48 -0600 (Mon, 04 Oct 2004)
Changed paths:
    M /sesame/trunk/timelib/Time.java
Added freeze/unfreeze time methods
------------------------------------------------------------------------
r13 | mike | 2004-10-04 21:10:50 -0600 (Mon, 04 Oct 2004)
Changed paths:
    M /sesame/trunk/timelib/Time.java
Added getCurrentDate() method
------------------------------------------------------------------------
```

■ 重命名目录

在 Subversion 中，目录和文件一样是一等公民。我们可以使用 svn move 命令很容易地重命名目录。可能时间库中有一些额外的辅助类添加了进来，它应该被重命名为 util。

```
timelib> cd ..
sesame> svn move timelib util
A         util
D         timelib/Clock.java
D         timelib
sesame> svn commit -m "Renamed timelib to util"
Deleting        timelib
Adding          util
Adding          util/Clock.java
Committed revision 16.
```

■ 使用项目仓库 URL

我们到现在所见过的 svn move 命令都是在工作拷贝中运行的。移动、改名、添加、删除这些操作都是在客户端先完成，再提交给服务器。在大部分情况下，这是合适的做法，因为程序代码经常需要编辑了之后才能让代码编译并通过测试。

Subversion 也让你使用项目仓库 URL 来运行这些命令，完全不需要工作拷贝。改动直接作用于项目仓库并需要一条提交消息。如果你有大量大文件并且不想使用工作拷贝来移动它们的时候，就应该使用这种方式。如果你要移动的是代码，就值得三思了。没有工作拷贝，你是没法运行测试的，从而很有可能会破坏什么东西。

要执行基于项目仓库的重命名，使用两个类似最开始签出时用的那样的 URL。让我们把 util 目录重命名为 common:

```
work> svn move -m "Renamed util to common" \
        svn://olio/sesame/trunk/util       \
        svn://olio/sesame/trunk/common
Committed revision 17.
```

回到 Sesame 工作拷贝中，执行一个更新可以得到最新的 common 目录并删除老的 util 的目录：

```
sesame> svn update
A  common
A  common/Clock.java
D  util
Updated to revision 17.
```

6.6 查看改变了什么

svn diff 命令向你展示文件版本之间的差异。你可以比较项目仓库中的文件与本地修改过的拷贝，也可以比较项目仓库中文件的两个版本之间的不同。

■ 查看你在工作拷贝中改动了什么

svn diff 的最简单的用法是查看你自从上次在项目仓库更新了工作拷贝之后都改动了什么：

```
common> svn diff Clock.java
Index: Clock.java
===================================================================
--- Clock.java (revision 21)
+++ Clock.java (working copy)
@@ -20,6 +20,11 @@
                frozen = true;
        }
+       public static void setTime(long time)
+       {
+               frozenTime = time;
+       }
+
        public static void unfreezeTime()
        {
                frozen = false;
```

这儿，可以看到我们最后把 Clock.java 文件更新到了版本 21，然后添加了 setTime()方法。

基本的 svn diff 命令列出的改动是你工作拷贝中的文件和你最后更新的版本的差别。Subversion 可以在不连接服务器的情况下完成这个任务，因为它储存了你工作拷贝中每个文件本地拷贝的之前版本。然而，如果其他人改动了文件并把他们的改动签入了项目仓库，你在差异分析结果中是看不见这些的。我们很快就能看到如何处理这种情况了。

■ 使用 Subversion 版本标识符

在签出和更新的时候，我们了解过 Subversion 的-r 选项。事实证明，在 Subversion 中，有很多地方我们都须要引用版本。你在-r 后面提供的选项称作**版本标识符**。当你使用版本标识符时，Subversion 接受版本号、日期和一些符号名，列于下表。

revision identifier

号	项目仓库的版本号，比如 87
{日期}	在日期开始时的版本，比如{“2004-09-26 13:35:06”}。花括号告诉 Subversion 你的使用日期，而且如果你使用的日期格式中包括空格的话，就需用引号括起来。Subversion 支持大量的日期格式，包括基本的 HH:mm，代表今天的某个特定时间。
HEAD	项目仓库中存储的最新版本。
BASE	工作拷贝的基准版本，即你最后签出或者更新到的那个版本。
COMMITTED	最后改动的版本，等于或者早于 BASE。
PREV	COMMITTED 之前的一个版本。

revision range

一些命令接受“**版本范围**”，这不过是用冒号分隔的两个版本标识符。版本范围用于引用两个时间上分隔的版本。

符号版本 BASE、COMMITTED 和 PREV 只能用于引用工作拷贝中的一个文件，因为在其他情况下没有意义。

图 6.1 展示了 Subversion 的符号版本。在本例中，你在工作拷贝中的 `Graph.java` 的版本是 2，另外一个开发者签入了一些改动，在项目仓库中创建了版本 3。因为你没有更新你的工作拷贝，这份 `Graph.java` 的版本是 2。PREV 版本是这个之前的一个版本，叫做版本 1。HEAD 总是项目仓库中最新的版本，在本例中是版本 3。

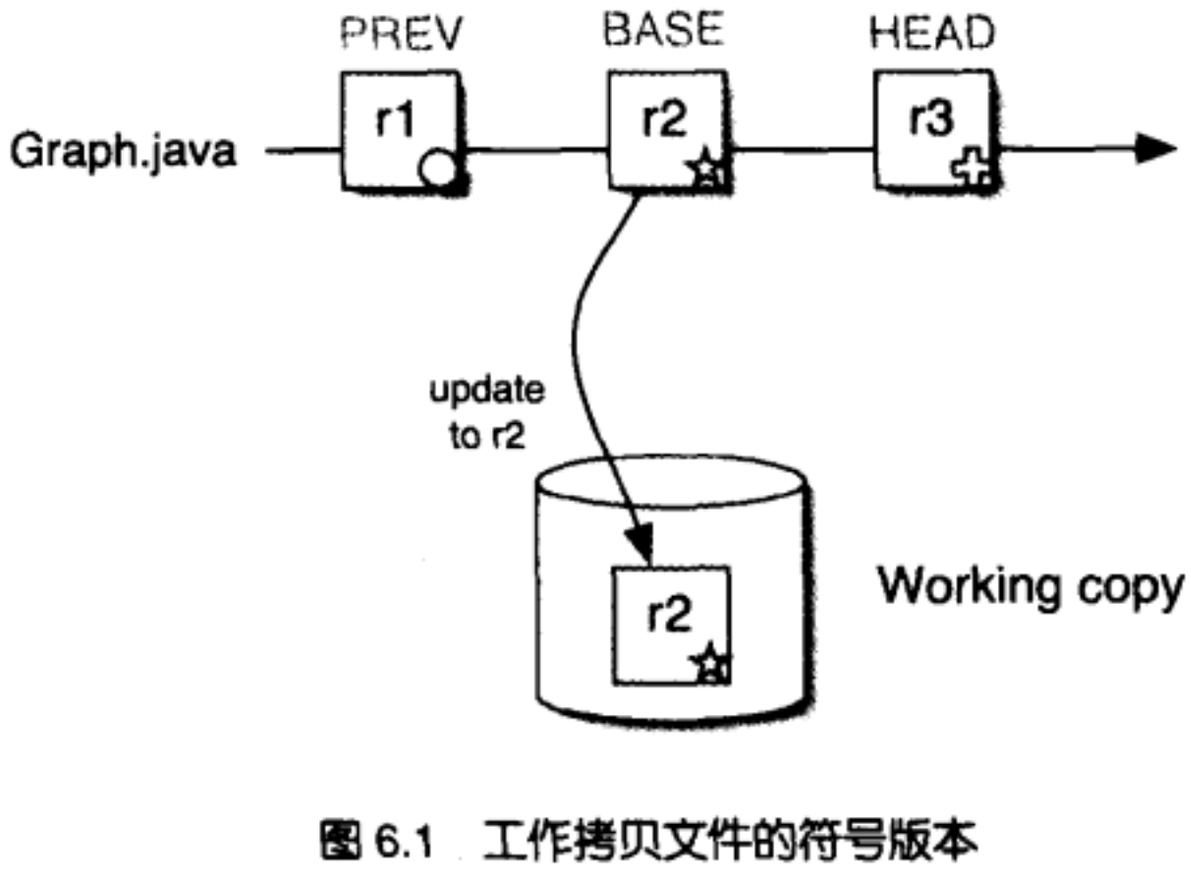


图 6.1 工作拷贝文件的符号版本

■ 在版本之间查找差异

查找指定文件版本之间的差别，使用-r 选项来指定一个版本范围：

```
common> svn diff -r19:21 Clock.java
Index: Clock.java
===================================================================
--- Clock.java (revision 19)
+++ Clock.java (revision 21)
@@ -1,9 +1,11 @@
  package timelib;
+import java.util.Date;
+
  public class Clock
  {
-     private boolean frozen = false;
-     private long frozenTime = 0;
+     private static boolean frozen = false;
+     private static long frozenTime = 0;
      public static Date getCurrentDate()
      {
```

即使工作拷贝没有包含任何历史信息也没有关系，这儿我们使用了工作拷贝中的一个文件来产生差异。在底层，Subversion 把文件路径翻译成项目仓库的 URL，因此它可以在需要的时候获得过去的版本。

如果你没有工作拷贝，可以直接在项目仓库上查找差异：

```
common> svn diff -r19:21 \
            svn://olio/sesame/trunk/common/Clock.java
Index: Clock.java
===================================================================
--- Clock.java (revision 19)
+++ Clock.java (revision 21)
@@ -1,9 +1,11 @@
  package timelib;
+import java.util.Date;
+
  public class Clock
  {
-    private boolean frozen = false;
-    private long frozenTime = 0;
+    private static boolean frozen = false;
+    private static long frozenTime = 0;
     public static Date getCurrentDate()
     {
```

前面我们注意到了使用 `svn diff` 时的一个常见错误，它不会显示项目仓库中改变了什么。要让 Subversion 去比较你的工作拷贝和项目仓库中的最新版本，使用 HEAD 关键字：

```
common> svn diff -r HEAD Clock.java
Index: Clock.java
===================================================================
--- Clock.java (revision 26)
+++ Clock.java (working copy)
@@ -1,6 +1,7 @@
  package timelib;
  import java.util.Date;
+import java.util.Calendar;
  public class Clock
  {
@@ -24,6 +25,11 @@
          frozenTime = System.currentTimeMillis();
      }
+    public static void switchToGMT()
+    {
+        frozenTime -= Calendar.getInstance()
+                                .get(Calendar.ZONE_OFFSET);
+    }
+
      public static void setTime(long time)
      {
          frozenTime = time;
@@ -38,10 +44,5 @@
      {
          frozen = false;
      }
-
-     public static boolean isFrozen()
-     {
-         return frozen;
-     }
  }
```

如果我们在工作拷贝中的 Clock 添加 switchToGMT()方法。同时，另外一个开发者添加了 isFrozen()方法并签入。当我们要求查看与 HEAD 之间的差异时，可以看到本地改动是额外添加的，而那个开发者的改动是“**要去掉的**”，也就是说，如果我们就拿工作拷贝中现在的方法去签入，就会回退添加 isFrozen()时所做的改动。

幸运的是，Subversion 不让这么做。要在更新了之后才能签入，这样 isFrozen()就会添加到我们的工作拷贝中来。

有时在你开始改动一个文件之前希望能够看到它最近的改动。你可以使用 PREV 符号版本做到这点：

```
common> svn diff -r PREV:BASE Clock.java
Index: Clock.java
===================================================================
--- Clock.java (revision 22)
+++ Clock.java (working copy)
@@ -26,6 +26,11 @@
         frozenTime = time;
     }
+   public static void setTime(Date date)
+   {
+         frozenTime = date.getTime();
+   }
+
    public static void unfreezeTime()
    {
         frozen = false;
```

这儿，Subversion 向我们展示之前对 Clock.java 的改动是添加了 setTime()方法。

Subversion 的 diff 命令还能用来查看不同开发分支之间的差异或者向你展示自从某个版本打了标签之后改变了什么。第 115 页的第 9 章“使用标签和分支”就是讲如何在不同标签版本和分支之间查找差异和合并的。

■ 查找差异和打补丁

如果你在开源社区里混过一段时间，你就应该认识那些拿着源代码补丁满世界晃悠的家伙。这些补丁与 Subversion 产生的差异报告是一样的东西，而且事实证明它们相当好使。

也许你正在使用开源的库，而且需要对其做某些修改。库是位于 CodeHaus 的[5]，它给开源开发者提供了很多东西，其中就包括免费的 Subversion 项目仓库。作为大众的一员，CodeHaus 让你从项目仓库中签出项目的源代码，但是因为你不在开发者的列表中，你就不能把改动签入。

这就是补丁能够派上用场的地方。只要让 Subversion 给你一个做过的改动列表（使用 `svn diff`）。把包含这个 diff 输出的文件用邮件发送给库的维护者，他就能使用补丁程序把这些补丁应用到源代码中。

下面的命令创建了一个称为 `mychanges.patch` 的文件，包含了对在 `oslibrary` 以及其下文件的所有改动：

```
oslibrary> svn diff > mychanges.patch
```

你可以把这个文件发送给维护者，他只要使用（惊讶吧！）`patch` 命令就能把这个补丁应用到他的源代码中去：

```
oslibrary> patch -p0 -i mychanges.patch
```

使用 `patch` 是一项高深的艺术，也许没法在本书这样的篇幅中传授，但是可以在这儿粗略地讲讲发生了什么：

- 补丁应用于`oslibrary`目录，就是创建它的“同一个目录”。
- `-p0`选项是告诉`patch`在应用它之前，从文件名中略过零层目录。如果你不包含这个选项，`patch`会抱怨说它找不到正确的文件。
- `-i`选项告诉`patch`使用`mychanges.patch`作为输入。

`patch` 是相当聪明的，一般都能够忽略掉补丁前后的“垃圾”文本，因此你把包含了他人改动的邮件保存下来，并整个地拿来用。大部分人会忘记

[5] `http://codehaus.org/`

魔法般的-p0，导致 patch 找不到正确的文件。

补丁在开源世界之外也是有用处的。你可以使用补丁把建议的改动发送给你项目团队的其他成员。如果你的客户有你的代码，你甚至可以使用补丁来分发那种三天两头凌晨三点急着要的 bug 修正代码。只要记住在你改动完之后同时签入到项目仓库中就行了。

6.7 处理合并冲突

Subversion 不加锁文件：[6] 项目中的所有人都能在任何时候修改任何文件。Subversion 的这个特性也许会让某些人夜不能寐。“什么能阻止两个人同时编辑同一个文件呢？”他们问。“做过的工作会不会白费？”

简单的答案是“不会，什么都不会。”如果你编辑了同一文件的不同部分，Subversion 会高兴地把两个改动合并到一起去，啥都不会影响。

然而，有时两个人会同时编辑文件的同一部分（虽然这种情况比你起初设想的要少得多）。当这样的事情发生之后，Subversion 不能自动执行合并，它不知道保留哪份改动。在这种情况下，Subversion 就会告诉你文件的两个版本产生了冲突，把麻烦甩给人（也就是你）来解决。

为了演示冲突的情景，我们要再用上我们的老朋友 Number.txt。这次，我们把它签出到两个不同的目录中。[7]

---

[6] 至少，Subverison 默认不对文件加锁。Subversion 1.2 支持可选的加锁措施，有时称为“保留签出”，在第 103 页的第 7 章“文件加锁和二进制文件”中我们会对此加以讨论。

[7] 眼尖的读者可能会注意到这些例子把我们带回到了项目仓库版本 1 的时代，那时我们只有两个文件，也没有 timelib 目录。这是可能的，通过一些管理 Subversion 的技巧——我们制作一个项目仓库的备份，只包括版本 1，把备份载入到一个新的项目仓库。第 176 页的 A.6 节“备份你的项目仓库”中对此有更详细的解释。

```
work> svn checkout svn://olio/sesame/trunk sesame1
A  sesame1/Number.txt
A  sesame1/Day.txt
Checked out revision 1.
work> svn checkout svn://olio/sesame/trunk sesame2
A  sesame2/Number.txt
A  sesame2/Day.txt
Checked out revision 1.
```

在 sesame1 目录中，我们会改变 Number.txt 的第一行，改动后它的内容如下：

```
ZERO
one
two
```

我们把这次改动签入：

```
sesame1> svn commit -m "Made zero uppercase"
Sending        Number.txt
Transmitting file data .
Committed revision 2.
```

现在来看 sesame2。记住我们想要创造合并冲突，所以假装不知道有人改动了要修改的文件。在 sesame2 中，改动 Number.txt，把第一行改成 Zero：

```
sesame2> svn commit -m "Capitalized 'Zero'"
Sending        Number.txt
Transmitting file data .svn: Commit failed (details follow):
svn: Out of date: '/sesame/trunk/Number.txt' in transaction '9'
```

到目前为止，一切正常。Subversion 检测到 Number.txt 过期了，所以执行一个 svn update：

```
sesame2> svn update
C  Number.txt
Updated to revision 2.
```

Subversion 以 C 标记文件，让我们知道在合并的时候有一个冲突，然后修复这个问题就是我们的事情了。

■ 修正冲突

修复合并冲突时要问的第一个问题是，“为什么会发生这样的事情？”这不是一个怪谁的问题，而是一个沟通的问题。为什么两个开发者会在同一时间编辑同一个文件的相同行？

有时，会有一个好理由来回答这个问题。可能他们同时发现了同一个 bug，而且都决定去修复它。或者可能他们都在添加功能时用到了通用的数

据结构，而且两人同时给那个结构添加了字段。这些是有道理的改动，但它们可能导致冲突。

但是，冲突的发生往往是因为人们应当让彼此知道哪些事情有人在做，但都沟通不够引起的。因此，我们强烈推荐，如果你遇到了一个合并冲突，又没有合理的解释，你就做一个记号提醒自己在下一次团队会议上把问题提出来。这样做的目的在于讨论问题的原因并找出改善交流的办法，以减少将来发生把人累死的事情。

现在一切正常，只是你要去面对这个冲突。Subversion 使用<<<和>>>字符序列把问题标记在文件的本地拷贝中了：

```
<<<<<<< .mine
Zero
=======
ZERO
>>>>>>> .r2
one
two
```

Number.txt

这儿可以看到我们的改动，Zero，很有帮助地被标记为 mine 了。还有来自项目仓库的改动，ZERO，有一个提示说这来自于版本 2。

我们现在需要决定如何修复这个问题。在真实世界中，你还得与做出这个改动的人去沟通；简单地抹掉别人辛苦的劳动成果，用你的版本去替换它，这样做很可能会让你在下次项目外出野餐时被扔下来没人理睬。

解决方案可能有很多种：

- 你决定扔掉你的改动，使用项目仓库中的版本。你要做的就是使用 svn revert命令，Subversion会删掉你的改动并使用项目仓库中的文件版本：

```
sesame2> svn revert Number.txt
Reverted 'Number.txt'
sesame2> svn update Number.txt
At revision 2.
```

- 决定保留你的改动并且丢掉项目仓库中那些东西。当冲突发生时，Subversion保存了每个版本的拷贝文件，后缀分别是.mine,.r1,.r2，等等。拷贝你的那个以.mine后缀结尾的版本文件，也就是到原始

冲突和花括号战争

假设两个开发者想要把他们的代码以不同的格式排版。Fred 喜欢他的代码用两个空格缩进，而且喜欢把所有的花括号放在声明的同一行。他的代码可能看起来是这个样子：

```
for (i = 0; i < max; i++) {
  if (values[i] < 0) {
    process(values[i]);
  }
}
```

然而，Wilma 喜欢她的代码用四个空格缩进而且不喜欢 Fred 的代码缩成一团的样子。她把她的花括号放在与声明不同的行。如果 Wilma 写同一块代码，它看起来应该是这样：

```
for (i = 0; i < max; i++)
{
    if (values[i] < 0)
    {
        process(values[i]);
    }
}
```

一天 Fred 在编辑 Wilma 的一些代码但不喜欢这种缩进方式。他操作编辑器重新缩进整个文件为两字符缩进的方式，并且把花括号放在他喜欢的位置。然后对一行做了一些小改动，存盘并且把改动提交到项目仓库。

问题是以 Subversion 的角度看来，文件的每一行都被改动了。如果 Wilma（或者其他任何人）改动了一些东西，他们会得到一个合并冲突，因为 Fred 对于缩进的改动意味着在项目仓库中的对应行与在 Wilma 的工作拷贝中的行不一样了。

现在你可以绕过这个问题：比如你可以告诉 Subversion 使用一个外部的 diff 程序，它能在判断文件之间的改动时忽略空白中的改动。然而，这并没有影响你确实修改了整个文件的事实，而且那些做了本地改动的人会在他们下次更新的时候遇到冲突。

冲突和花括号战争（续）

规则很简单：不要恣意改动共享文件的版式。如果你必须要改动缩进，首先须确信团队中的其他人没有在本地修改这个文件。然后改动排版并签入改动过的文件，其他什么都不要动。然后告诉大家去更新，因此他们会在新的版本上开发。这会减少人们遇到冲突的次数，也会减少你收到不悦信件的数量。

文件，并且告诉Subversion你已经修正好冲突了：

```
sesame2> cp Number.txt.mine Number.txt
sesame2> svn resolved Number.txt
Resolved conflicted state of 'Number.txt'
```

当你告诉它已经修正好冲突时，Subversion会清理掉所有.mine和.r2文件。

- 如果你决定想要多个版本的样式都用一点，那么你就需要做一些手工编辑了。编辑包含了冲突标记的文件，把它修改成你想要的样子。确保你把冲突标记给移除了。举个例子，在我们的例子中可能决定第一行不应该是Zero也不应该是ZERO而应该是Empty：

```
<<<<<<< .mine                                  Empty
Zero                                           one
=======                        becomes =>      two
ZERO
>>>>>>> .r2
one
two
```

Subversion 不会让你提交一个仍然处于冲突状态的文件的。[8]为了让

[8] 对于项目很大、文件很多的同志来说，这下他们可以放心了。“要是因为卷屏了，我看不到文件边上的 C 怎么办？”这是一个很常见的问题。如果你漏掉了一个冲突，而且代码中还有大量的<<<字符，这不会很可怕地对你的持续集成造成破坏，因为当你尝试提交的时候，Subversion 会抱怨说有文件还是处于冲突未解决的状态。

Subversion 知道你修正了冲突，使用 `svn resolved` 命令：

```
sesame2> svn resolved Number.txt
Resolved conflicted state of 'Number.txt'
```

6.8 提交改动

在我们做了一系列改动之后（而且，理想情况下，你还运行过所有的测试并且没有破坏任何东西之后），你想要把它们存储到项目仓库中。在本书中我们已经做过好多次了；你只需要使用 `svn commit`。

然而，我们想要推荐一个稍微复杂一点的命令组合供每次提交的时候使用：

```
myproject> svn update
myproject> #... resolve conflicts ...
myproject> #... run tests ...
myproject> svn commit -m "check in message"
```

第一行让我们的本地工作拷贝更新到项目仓库的当前状态。这很重要，因为虽然我们的代码与上次更新时获得的项目的其他文件工作起来一切正常，但是其他人可能改动了一些东西让新的代码没法正常工作。在更新之后，我们可能还要解决冲突。

即使没有冲突存在，我们也应该再次编译和测试代码，修复所有遇到的问题。这保证了当我们签入的是在更大的项目上下文中可以正常工作的代码。你需要一个很快的测试集以使得开发者可以用这种方式工作，人们是不会喜欢在他们的测试运行时等上多于数分钟的时间的。

一旦我们检查过所有东西都是正确的了，就可以提交我们的改动，使用 `-m` 选项添加一个有意义的消息。如果你忽略了 `-m` 选项，Subversion 会打开编辑器让你输入更长一些的注释。

6.9 查看改动历史

你可以使用 `svn log` 命令查看你和你的团队成员输入过的日志消息：

有意义的日志消息

怎样的日志消息算是好的日志消息？为了回答这个问题，假定你是另外一个开发人员，在数年之后来阅读现在的代码。你对于系统的某些部分不是很明白，尝试着弄清楚有些东西为什么会用这种方式来做。你注意到改动是在这一块，而且希望日志消息会给你一些提示，让你知道某种设计决定的动机是什么。

现在，回到眼前来。你几天在日志消息上撒些什么样的“面包屑”能在数年之后帮助你的同事呢？

答案部分来源于意识到 Subversion 已经储存了你对代码改动的细节。在日志中写“把 timeout 改为 42。”是没有意义的，因为用一下 diff 就能看出来 `setTimeout(10)` 变成了 `setTimeout(42)`。应该用日志消息来回答“为什么要这么做？”这样的问题：

```
If the round-robin DNS returns a machine that
is unavailable, the connect() method attempts
to retry for 30mS. In these circumstances our
timeout was too low.
```

如果改动是为了完成一个 bug 报告，把跟踪号写到日志消息中：问题的描述已经在 bug 数据库中了，无须在这儿再重复。

```
sesame> svn log Number.txt
------------------------------------------------------------
r4 | mike | 2004-09-08 22:45:16 -0600 (Wed, 08 Sep 2004)
Make 'six' important
------------------------------------------------------------
r3 | mike | 2004-09-08 22:05:32 -0600 (Wed, 08 Sep 2004)
Customer wants more numbers
------------------------------------------------------------
r1 | mike | 2004-09-08 21:50:13 -0600 (Wed, 08 Sep 2004)
------------------------------------------------------------
```

如果你只是想了解一下谁最近改动过，你可以让 Subversion 给你某个目录中发生过的一切日志。在一棵大的目录树的顶端执行这项操作可能会产生大量的输出，可以通过 more 命令使用管道[9]来分页 Subversion 的输出：

```
work> svn log sesame | more
```

Subversion 接受-r 选项来指定你关注的版本。使用单独一个版本号会显示那个版本改动了什么，使用一个版本范围会显示那一段时间的历史：

```
sesame> svn log -r 19:24 Clock.java
------------------------------------------------------------
r19 | mike | 2004-10-04 21:47:09 -0600 (Mon, 04 Oct 2004)
Renamed util to common
------------------------------------------------------------
r21 | mike | 2004-10-09 16:33:00 -0600 (Sat, 09 Oct 2004)
Fixed compilation problems
------------------------------------------------------------
r22 | dave | 2004-10-09 16:48:23 -0600 (Sat, 09 Oct 2004)
Added setTime() method
------------------------------------------------------------
r23 | ian | 2004-10-09 17:00:23 -0600 (Sat, 09 Oct 2004)
Added setTime() method taking a Date
------------------------------------------------------------
r24 | dave | 2004-10-10 18:07:08 -0600 (Sun, 10 Oct 2004)
Added Log class
------------------------------------------------------------
```

这儿我们要求查看 Clock.java 从版本 19 到版本 24 的改动。我们在项目仓库的第 20 个版本中实际上并没有改动过 Clock.java，这也就是为什么在这儿少了一个版本的原因。还要注意 Subversion 是如何打印版本的，最新的版本列在最后面。如果没有使用-r 选项获得的日志，则会把最新的版本放在最上面。

最后一个消息说“Added Log class”，但是却作为 Clock.java 的历史的一部分显示出来。这看起来有一点奇怪，所以让我们使用-v（verbose，冗余）

[9] 在美国键盘上管道字符是 Shift+\ 。

选项获得更多信息：

```
common> svn log -r 24 -v Clock.java
------------------------------------------------------------------------
r24 | dave | 2004-10-10 18:07:08 -0600 (Sun, 10 Oct 2004)
Changed paths:
   M /sesame/trunk/common/Clock.java
   A /sesame/trunk/common/Log.java
Added Log class
------------------------------------------------------------------------
```

现在 Subversion 给出的内容更多了，我们可以看到版本 24 添加了 `Log.java` 还改动了 `Clock.java` 文件。在本例中，新的 `Log` 类依赖于 `Clock` 的一些额外功能。因此，当 Dave 提交他的改动时同时提交了两者，从逻辑上是说得通的。

Subversion 在单次提交中可以跟踪多个文件的改动的能力是极其强大的。如果你浏览某个特定文件的历史，并且看到了一个值得你注意的改动，添加`-v` 选项到 `svn log` 命令以显示在那次提交中改动的所有文件。比如，在你需要找出要改哪些东西才能修正某个 bug 时会很好使。

■ 逐行历史

`svn blame`[10]命令显示一个或者多个文件的内容。对于每个文件中的每一行，它会显示改动了那行的最后的版本，以及做出这个改动的人：

```
sesame> svn blame Number.txt
     10       mike # $HeadURL$
     10       mike # $Id$
      5       dave ZERO
      7       mike ichi
      7       mike due
      1       mike three
      1       mike four
      3       andy five
      4       ian SIX
```

这是一个很好的工具，当你被卷入到软件考古的活动中时；你可以快速地找出改动的模式以及辨别出某个特定版本修改了哪些行代码。

---

[10] 它被称为 blame，因为它经常用来决定谁对某一块代码负责（或者某个 bug！）。`svn blame`，`praise`，`annotate` 以及 `ann` 都是相同的东西。

svn blame 接受-r 选项来指定版本或者版本范围。这样的话，Subversion 在显示逐行历史时就不会查看文件的所有历史了。

6.10 移除改动

有时我会想丢弃掉对代码做过的某些改动。

如果改动是在本地未签入的工作拷贝中，那么我们使用 svn revert 就能把改动给扔掉。

给 CVS 用户的提示：CVS 用户习惯于只把本地改动过的文件删除，然后执行一次更新来还原文件。在 Subversion 中也是可以这么用的，但是执行 revert 实际上更加安全也更迅速。更新会去连接服务器并且可能取回你还没有准备好接受的新的改动，而 revert 不会连接服务器并且也不会从项目仓库中获取新的改动。

如果改动已经提交了，Subversion 能够帮助我们来移除它。做这件事有很多种办法；这儿要讲的是一种我们认为最简单也最不可能出错的做法。在本例中，让我们假设开发一个联系人管理系统。我们发布了一个早期版本到公测站点上，事情似乎很顺利直到团队在野餐的某一天，客户打电话过来说，当他们从他们的地址列表中移除一个联系人的时候，程序把所有的客户信息都移除了。

第一步是保证我们拿到的是最新的代码。

```
contacts> svn update
U  Contacts.java
Updated to revision 28.
```

然后找出要移除的确切版本。干这事用 svn log 很适合。让我们来看看主要的管理联系人的类的日志吧：

```
contacts> svn log Contacts.java
------------------------------------------------------------------------
r28 | mike | 2004-10-11 10:54:08 -0600 (Mon, 11 Oct 2004)
Reformat PMB Addresses
------------------------------------------------------------------------
r27 | fred | 2004-10-11 10:52:47 -0600 (Mon, 11 Oct 2004)
Remove from database too
------------------------------------------------------------------------
r26 | ian | 2004-10-11 10:51:38 -0600 (Mon, 11 Oct 2004)
Sort clients into alpha order (Bug 2942)
------------------------------------------------------------------------
```

版本 27 看起来值得怀疑，因此我们使用 `svn diff` 来看到底在版本 26 和版本 27 之间改动了什么：

```
contacts> svn diff -r 26:27 Contacts.java
Index: Contacts.java
===================================================================
--- Contacts.java    (revision 26)
+++ Contacts.java    (revision 27)
@@ -25,6 +25,7 @@
     public void removeClient(Client client)
     {
+        database.deleteAll(client);
         clientList.remove(client);
     }
}
```

这看起来就是那个问题。然而，在开始 hack 其他人的改动之前，让我们先做一些调查。查看日志，我们可以看到这行是 Fred 改的，因此去询问一下此事。事实证明这只是一个误解；Fred 没有意识到这个调用会删除客户的所有记录。移除这个改动没有问题。[11]

现在我们必须移除版本 27 中对 `Contacts.java` 的改动。可使用 `svn merge` 命令撤销改动：

```
contacts> svn merge -r 27:26 Contacts.java
U Contacts.java
```

我们让 Subversion 计算 `Contacts.java` 版本 27 和 26 之间的差异，并把改动应用到工作拷贝中。我们使用的是 27:26，因为可以使用 `svn diff` 来验证 Subversion 确实正确地撤销了改动：

```
contacts> svn diff Contacts.java
Index: Contacts.java
===================================================================
--- Contacts.java    (revision 28)
+++ Contacts.java    (working copy)
@@ -26,7 +26,6 @@
     public void removeClient(Client client)
     {
-        database.deleteAll(client);
         clientList.remove(client);
     }
```

[11] 审慎的话，还要快速地搜索其余的代码看看 Fred 是否在其他地方使用了 `deleteAll()`。

此时，回到正常的流程中了。我们对源代码做了一个改动，所以应该测试它并且把改动提交到项目仓库：

```
contacts> svn commit -m "Revert deleteAll change from r27"
Sending        contacts/Contacts.java
Transmitting file data .
Committed revision 29.
```

■ 撤销更大的改动

我们刚刚看到的例子是撤销一个文件的改动。我们如何处理牵涉多个文件的改动呢？

幸运的是，Subversion 在每次提交时跟踪了我们改动的所有文件；只要改动是按逻辑分组好的，要撤销它们就很容易。如果 r27 改动了 contacts 目录下的好几个文件，我们可以通过使用“.”（当前目录）作为目标来撤销所有的改动：

```
contacts> svn merge -r 27:26 .
U Contacts.java
U Database.java
```

把相关的改动合在一起作为一个版本提交是很重要的。如果单个逻辑上的改动，比如说“添加生日字段”，分布在好几次提交中，要撤销改动就更加困难，而且也更难跟踪哪些文件改过了。当浏览历史的时候，你可以使用 -v（verbose，冗余）选项列出具体版本中改动过的所有文件。

svn merge 命令还可以让你在合并的时候指定项目仓库 URL。第 115 页的第 9 章“使用标签和分支”中，我们使用这个来在分支之间合并改动。

■ 检查你的工作拷贝

在你的工作拷贝上做开发、编辑文件和添加新文件（偶尔也会删除文件）。与此同时，团队中的其他成员也在做相同的事情，他们也在把他们的改动签入到项目仓库中去。结果就很容易失去对工作拷贝的状态的跟踪。特别是会经常忘记把你工作拷贝中的新文件添加到项目仓库中去。

svn status 命令可以得到你工作目录中文件的信息：

```
proj> svn status
?       common/Calendar.java
M       contacts/Contacts.java
```

这儿 Subversion 告诉我们，工作目录中的 Calendar.java 没有被添加到版本控制系统之中，同时还可以发现我们改动了 Contacts.java。

默认情况下 Subversion 只会显示工作拷贝的信息，而得到这些信息是无须访问网络的。如果其他人改动了项目仓库中的文件，我们不会知道拷贝过期了。然而，如果你指定--show-updates 选项的话，Subversion 会与服务器通信显示额外的信息（如果你想避免 RSI 的话可以使用-u）：

```
proj> svn status --show-updates
?                       common/Calendar.java
        *           26  common/Log.java
M       *           27  contacts/Contacts.java
Status against revision:      30
```

现在我们知道工作拷贝中的 Log.java 和 Contacts.java 都过期了。我们有版本 26 的 Log.java 和版本 27 的 Contacts.java（这个我们也修改了）。项目仓库当前的版本是 30，而且当我们去更新的时候，这些额外的改动会合并到工作拷贝之中。

PDG

第 7 章

文件加锁和二进制文件

File Locking and Binary Files

学习版本控制时遇到的常见问题是，“如果两个人同时编辑同一个文件会发生什么？我们会不会把时间浪费在无意义的事情上，每天都在撤销其他人的改动？”由于合并文本的魔力，大部分时间都不是问题。但是如果文件是图片或者 CAD 这种不可合并的文件呢？Subversion 1.2 引入了可选的文件加锁，它可以帮助避免由不可合并的文件导致的问题。

7.1 文件加锁概览

许多项目包含不可合并的文件。声音、图片以及更多文档格式不可以以有意义的方式合并。如果 Alice 和 Bob 都决定“同时”编辑 `CurrencyConversionRates.xls`，他们当中的一个会先提交，另外一个就必须重做他做过的改动了。

从根本上来说，这个问题是因为你的团队没有沟通好。应该先给团队的其他人打句招呼，问问他们是不是打开了那个文件，因为这可能会浪费他们的时间。Alice 和 Bob 两个人同时都要编辑项目的主题歌文件，这样的情况是不大可能发生的。Subversion 提供了一种机制通过可选的**文件加锁**来帮助团队沟通。

file locking

任何文件都可以被设置为加锁模式，这样的话，在编辑之前就需先开锁。设置加锁模式是通过设置它的 `needs-lock` 属性（属性包含什么没有关系。

只要它被设置了，Subversion 就会对那个文件加锁）。任何加锁了的文件签出到工作拷贝中的时候都是只读的。大部分现代的编辑器都会拒绝编辑只读的文件，至少也会在你编辑它之前发出警告，提示你说文件需要先开锁才能编辑。

lock token

我们可以使用 `svn lock` 命令来获得文件的锁。Subversion 客户端会与服务器通信确保文件不是已经被锁住了，获得一个“**锁令牌**”，然后在工作拷贝中把文件标记为可读可写。另外我们可以指定一个加锁注释告诉其他用户我们为什么要锁住这个文件。

如果一个文件被锁住了，另外一个用户在砸坏原先的锁之前不能锁住这个文件或者提交改动到这个文件（后面我们会对什么情况下这是合适的有更多讨论）。当锁住文件的用户把事情做完了，提交了他的改动，锁就被解开了。

让我们来看看这在实际操作中是怎么做的吧。如果你想要跟着这些例子边看边做，为 Alice 和 Bob 创建一个单独的工作拷贝，类似于第 89 页的 6.7 节“处理合并冲突”中我们创建过的那样。

7.2 实战文件加锁

Sesame 项目在向全球拓展。现在我们的客户希望软件在它的所有现有功能之外，在其他国家也能运行。以此为动机，我们要在不同的当地货币之间进行转换。真实系统会使用某种 web 服务来获取兑换汇率，但是测试时，Bob 想要使用像 Excel 电子表格这样简单的东西。

■ 为什么文件加锁至关重要

Bob 创建了一个电子表格文件，`CurrencyConversionRates.xls`，并且把它添加到了项目仓库之中：

```
sesame>svn add CurrencyConversionRates.xls
A (bin) CurrencyConversionRates.xls
sesame>svn commit -m "Added conversion rates for testing"
Adding (bin) CurrencyConversionRates.xls
Transmitting file data .
Committed revision 32.
```

Subversion 在添加的时候会自动检测到电子表格是二进制文件，同时会把它标记为不可合并的。如果 Alice 签出了版本 32 而 Bob 和 Alice 都改动了文件并且尝试去提交，他们中只有一个会成功。以下是 Bob 可能看到的：

```
sesame>svn commit -m "Added Norwegian Krona conversion rate"
Sending        CurrencyConversionRates.xls
Transmitting file data .svn: Commit failed (details follow):
svn: Out of date: '/trunk/CurrencyConversionRates.xls'
     in transaction '43-1'
```

Bob 的工作拷贝是过期的，因为 Alice 先把她的改动签入了。如果这是一个文本文件，Bob 只须更新他的工作拷贝，Subversion 会合并 Alice 提交的和 Bob 正要提交的改动。然而对于电子表格这行不通；于是就产生了一个冲突：

```
sesame>svn up
C    CurrencyConversionRates.xls
Updated to revision 33.
```

Subversion 告诉 Bob，他的那份 `CurrencyConversionRates.xls` 与项目仓库中的新版本发生了冲突。Bob 现在有几种选择。他可以使用 `svn log` 做一些研究，看看谁改动了这个文件，他可以选择保留他的改动，保留 Alice 的改动，或者手工合并两者。无论怎么做都需要大量的工作。

■ 开启文件加锁

Bob 决定要扔掉他的改动并且重做。毕竟，他只是改动了电子表格中的一行，而且可以快速地把他的改动在最新版本上再做一遍。但是他想要避免将来再发生这样的问题，因此他给电子表格添加了 `svn:needs-lock` 属性。

```
sesame>svn propset svn:needs-lock true CurrencyConversionRates.
xls
property 'svn:needs-lock' set on 'CurrencyConversionRates.xls'
sesame>svn commit -m "Enabled locking for spreadsheet"
Sending        CurrencyConversionRates.xls
Committed revision 34.
```

Bob 把 `svn:needs-lock` 设置为“true”（记住属性的值是什么其实没有关系；只要它存在 Subversion 就会对这个文件加锁）。对这个属性的改动直到它被提交到了项目仓库之中才会生效。如果你正在添加一个不可合并的文件并且想要启用加锁，立马就设置好 `svn:needs-lock` 属性是一个好办法。

第 77 页的 6.4 节“自动设置属性”讲过的 Subversion 的 autoprops 对此会有帮助。

■ 基本文件加锁

一旦 Alice 和 Bob 更新了他们的工作拷贝，Subversion 会把 `CurrencyConversionRates.xls` 文件变成只读。把工作拷贝变成只读的背后想法是下次用户编辑这个文件的时候，他们可能会被提示正在修改的文件是只读的，并且会记住在继续之前把文件先锁上。有否提示取决于你用来编辑文件的程序，你可能会得到一个警告，也有可能不会。Excel 打开只读的文件没啥反应，不给你任何警告就让你改动它。只有等到你去保存改动过的电子表格的时候，你才会被提示说要输入一个新的文件名。虽然在大多数情况下这不是什么大问题，因为大部分用户直觉上会经常点击“保存”。其他的程序像图片或者声音编辑器可能对待只读文件会有所不同。你必须试试你用的程序才能知道它是怎么做的。

在设置了 `svn:needs-lock` 之后，Bob 决定再添加 Norwegian Krona 的兑换比率。这次，他在编辑这个文件之前先把它锁住。[1]

```
sesame>svn lock CurrencyConversionRates.xls
         -m "Adding Norwegian Krona"
'CurrencyConversionRates.xls' locked by user 'bob'.
```

在你锁住文件的同时值得推荐的做法是再包含一点注释。Subversion 可以给其他用户提供加锁注释，而且这是增强交流的好办法。Bob 现在可以查看文件，看看它是不是被锁住了：

```
sesame>svn info CurrencyConversionRates.xls
Path: CurrencyConversionRates.xls
Name: CurrencyConversionRates.xls
URL: svn://olio/sesame/trunk/CurrencyConversionRates.xls
Repository Root: svn://olio/sesame
Repository UUID: 63a31077-dc47-8e48-8372-099aabc6682c
Revision: 34
Node Kind: file
Schedule: normal
Last Changed Author: bob
```

---

[1] Subversion 只会在文件是更新到最新版本的情况下才会让你锁住这个文件，你不能锁住旧版的文件。

```
Last Changed Rev: 34
Last Changed Date: 2006-03-06 15:31:04 -0700 (Mon, 06 Mar 2006)
Text Last Updated: 2006-03-06 15:29:58 -0700 (Mon, 06 Mar 2006)
Properties Last Updated: 2006-03-06 15:30:40 -0700 (Mon, 06 Mar 2006)
Checksum: 7cd95b6dcf6b3ce39baf073f56253e20
Lock Token: opaquelocktoken:42eef8ec-0355-d548-805b-16b5a8e830aa
Lock Owner: bob
Lock Created: 2006-03-06 17:10:17 -0700 (Mon, 06 Mar 2006)
Lock Comment (1 line):
Adding Norwegian Krona
```

这儿有很多信息，但是我们所关注的内容是最后五行。可以看到 Bob 的工作拷贝有一个“锁令牌”，而且 Bob 是锁的拥有者。我们还可以看到当锁创建的时候，Bob 的注释解释了为什么他要锁住这个文件。

如果 Alice 现在尝试着去锁这个文件，她会得到一个错误提示。

```
sesame>svn lock CurrencyConversionRates.xls \
          -m "Adding Euro conversion rate"
svn: warning: Path '/trunk/CurrencyConversionRates.xls'
     is already locked by user 'bob'
     in filesystem '/home/svnroot/sesame/db'
```

Subversion 会让她知道 Bob 已经锁住了文件。如果 Alice 在她的工作拷贝运行 `svn info`，她不能看到锁令牌，表示她不拥有一个锁（她不能，Bob 已经有这个锁了）。Alice 要找出关于这个文件为什么被锁住的原因，她可以直接去问 Bob（改善团队交流）或者她可以询问 Subversion 服务器。以文件的完整 URL 去运行 `svn info` 能得到更多信息：

```
sesame>svn info svn://olio/sesame/trunk/Currency Conversion
       Rates.xls
Path: CurrencyConversionRates.xls
Name: CurrencyConversionRates.xls
URL: svn://olio/sesame/trunk/CurrencyConversionRates.xls
Repository Root: svn://olio/sesame
Repository UUID: 63a31077-dc47-8e48-8372-099aabc6682c
Revision: 34
Node Kind: file
Last Changed Author: bob
Last Changed Rev: 34
Last Changed Date: 2006-03-06 15:31:04 -0700 (Mon, 06 Mar 2006)
Lock Token: opaquelocktoken:42eef8ec-0355-d548-805b-16b5a8e830aa
Lock Owner: bob
Lock Created: 2006-03-06 17:10:17 -0700 (Mon, 06 Mar 2006)
Lock Comment (1 line):
Adding Norwegian Krona
```

Alice 需要使用完整 URL 才能找出关于 Bob 锁的信息。如果她只是使用文件名，Subversion 告诉的信息是关于她的工作拷贝的。Alice 的工作拷贝

并没有那个锁，所以她需要询问服务器才能知道文件的最新信息。

一旦 Bob 编辑完文件，他就可以提交了。当你提交一个文件或者目录的时候，Subversion 会自动解开任何你拥有的锁。[2]

■ 把锁砸坏

锁的拥有者可以使用 `svn unlock` 去打开它。但是如果锁的拥有者不在怎么办？假设 Alice 尝试着锁住兑换汇率文件以添加一些数据。Subversion 警告她文件已经被锁住了，所以她做了一些调查研究：

```
sesame>svn info \
svn://olio/sesame/trunk/CurrencyConversionRates.xls | grep
      Lock
Lock Token: opaquelocktoken:42eef8ec-0355-d548-805b-16b5a8e830aa
Lock Owner: bob
Lock Created: 2006-03-06 17:10:17 -0700 (Mon, 06 Mar 2006)
Lock Comment (1 line):
```

Alice 可以看到 Bob 锁住了这个文件，但是他昨天就锁住了，到现在仍然没有解开锁。因为 Bob 今天休病假，Alice 决定破坏掉这个锁做她可以做的改动。她决定留言提醒 Bob 以后不要把重要文件锁住太久。

`svn unlock` 命令可以用来解开别人对文件的锁。Alice 需要传入 `--force` 选项因为她自己并不拥有这个锁：

```
sesame>svn unlock svn://olio/sesame/trunk/CurrencyConversion
       Rates.xls
svn: warning: User 'alice' is trying to use a lock owned
     by 'bob' in filesystem '/home/svnroot/sesame/db'
sesame>svn unlock --force \
          svn://olio/sesame/trunk/CurrencyConversionRates.xls
'CurrencyConversionRates.xls' unlocked.
```

如果 Alice 忘记告诉 Bob 她破坏了他的锁，当 Bob 尝试着提交改动的时候，Subversion 会让他知道他不再拥有对应的锁令牌了。在 Bob 的工作拷贝中的令牌对应于 Alice 破坏掉的那个锁，因此 Bob 必须试着再次锁住这个文件，才能提交。

---

[2] Subversion 会解开你工作拷贝中的“所有的”文件的锁，而不仅仅是你提交的那些。这鼓励用户尽快地解开锁，但是可能在你头几次用这个功能时会遇到些麻烦。

```
sesame>svn commit -m "Added Norwegian Krona"
Sending        CurrencyConversionRates.xls
Transmitting file data .svn: Commit failed (details follow):
svn: Cannot verify lock on path '/trunk/CurrencyConversion
     Rates.xls';
     no matching lock-token available
```

看到 Alice 可以强制地解开文件的锁，你可能认为 Subversion 的文件加锁没啥作用。如果完全没有加锁的话，Bob 处于同样的状况，他必须决定是抛掉自己的改动还是覆盖掉 Alice 的。然而我们做到的是编辑文件的人之间的更好交流。Alice 知道她在破坏 Bob 的锁，而且有理由这么做：Bob 今天不上班而 Alice 需要继续工作。

Subversion 允许你限制谁能加锁和解锁文件，以及谁能破坏这些锁，通过使用特殊的钩子脚本。在文件加锁和解锁（分别地）之前会运行“*pre-lock*”和“*pre-unlock*”钩子。这些钩子可以查看文件是否已经被锁住了，可以根据公司实际的政策，把破坏锁的操作约束为只有特定用户才能执行。

还可以使用“*post-lock*”和“*post-unlock*”钩子，比如说可以在锁被破坏了之后发送电子邮件。这样 Alice 就不会忘记告诉 Bob 她破坏了他的锁了，因为有一封信等着让他知道这件事呢。

在强制解开了 Bob 的锁之后，Alice 应该锁住这个电子表格，从而她可以修改它。但是在 Alice 输入这两个命令之间，有很小几率其他人锁住了这个文件，因此 Subversion 还提供了从别的用户那儿把锁“偷”来的能力。

连同`--force`选项使用`svn lock`会偷来锁，不给任何其他人锁住这个文件的机会。

```
sesame>svn lock --force CurrencyConversionRates.xls
'CurrencyConversionRates.xls' locked by user 'alice'.
sesame>svn info CurrencyConversionRates.xls | grep Lock
Lock Token: opaquelocktoken:b8c434ce-98ef-1046-a553-7479
            abccdbca
Lock Owner: alice
Lock Created: 2006-03-07 14:44:26 -0700 (Tue, 07 Mar 2006)
```

值得注意的一点是，锁是对应于工作拷贝以及被具体用户所拥有的。如果 Bob 用他办公室的电脑锁住了文件，然后第二天他在家里用自己的笔记本工作的时候，锁仍然存储在办公室的机器上。如果他想在家编辑这个文件，他

必须要破坏这个由办公室的工作拷贝所持有的锁。

7.3 何时使用加锁

Subversion 的可选加锁对于控制访问不可合并的文件是非常有用的。添加 `svn:needs-lock` 给文件，可以帮助你的团队对谁在开发哪个文件这个问题上会更有效地沟通，并且可以帮助避免工作上的浪费。

每次尽可能快尽可能少地锁住文件。不要像 Bob 一样，锁住一个文件然后回家去过一夜。文件被锁住的时间越长，其他人等着修改这个文件的几率就越高。在最坏的情况下，Alice 可能要等着 Bob 完成一个文件的工作，而 Bob 要等着 Alice 完成另外一个文件的工作。他们两个人都要无止尽地等待另一个人的锁被解开。从诸如 Visual Source Safe 这样的系统迁移到 Subversion 来的开发者对于这样“死锁”的情况一定不陌生。

如果你的文件是文本，比如说程序代码，Subversion 通常可以为你合并改动，而且你无须锁住它们。在有些情况下，锁住可合并的文件看起来很吸引人，比如说某个重要的源代码文件，开发者频繁地要去更新，并且看起来遇到了大量的冲突。通常最佳的解决方案不是开始对这个文件加锁，而是去找出如何把文件分成数个逻辑单元，从而整个团队在改动的时候不需要持续地走来走去询问情况。

第 8 章

组织你的项目仓库

Organizing Your Repository

当使用版本控制系统的时候，你在大部分情况下都要存储一个以上的项目。一个 Subversion 项目仓库可以用来储存整个组织的所有程序都要使用的文件，无论这些开发者是不是在开发同一个项目。不同版本控制系统有着各自不相同的技术来把项目仓库分隔给不同的项目、子项目、模块、等等。Subversion 使用了相当简单的机制，就是把所有东西都组织到目录中去。

8.1 简单的项目

Sesame 项目是整本书中使用的主要例子。回到第 35 页的 3.3 节“创建简单的项目”中，我们把 Sesame 项目导入到了项目仓库的`/sesame/trunk`位置。那时我们没有讲为什么需要导入到 `trunk` 而不是把文件直接放在 `sesame` 目录下，现在是解释这个问题的时候了。

大部分项目会有一条主线开发，大部分开发工作都是在那里完成的。项目还可能有**发布分支**，那里的代码是完成了的，是在发布上线使用后存储在那里的。发布分支不会经常被改动，除了需要修正 bug 的时候。最后，项目生命周期中的重大事件经常要使用“**标签**”来记录。比如说，一个标签可能包含的代码是 Sesame 发布的第五版的代码。

main line

release branches

tags

第 115 页的第 9 章“使用标签和分支”有大量的内容讲标签和分支，现在你只须知道标签和分支都是通过拷贝 Subversion 项目仓库中的目录来创建的。对于标签，推荐的位置是 tags/ 目录。对于分支，则是 branches/ 目录。

为了让目录都能够很容易被找到，我们最后使用/sesame/trunk 作为主要的开发区域，把/sesame/tags 用来储存标签，以及/sesame/branches 用来储存分支。图 8.1 更直观地展现了这样的结构。

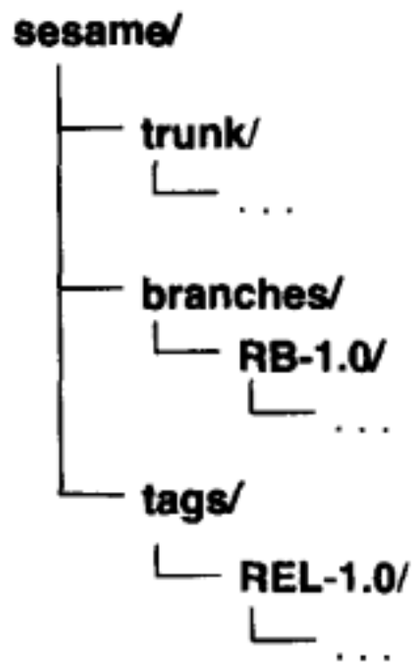


图 8.1 Sesame 项目的主干和分支

把项目代码存储在 trunk 目录中对应于 SCM 的“mainline”模式。

8.2 多个项目

到目前为止，我们有了一个储存着 Sesame 项目的项目仓库。不用说都知道如何储存其他项目了，比如说 Aladdin 和 Rapunzel，如图 8.2 所示。

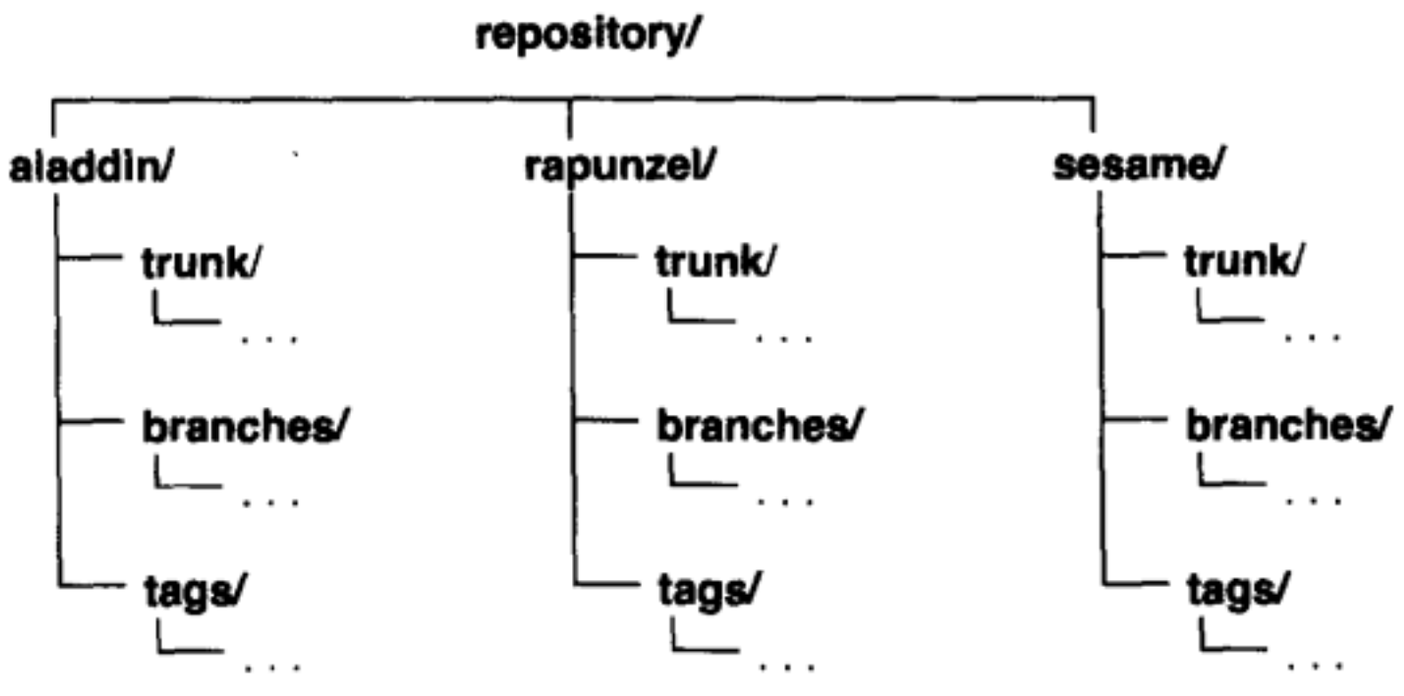


图 8.2 Aladdin 和 Rapunzel 项目

很重要的一点是 Subversion 使用目录拷贝来做分支和打标签，所以你不一定必须把你的标签目录命名为 tags。然而如果你使用的是不同的名字话，你的用户可能会一时找不到。你不一定必须把你的主干、标签、分支目录放在同一个目录下。也不一定要把每个项目放在项目仓库的根目录下——取决于你的开发者和 IT 部门是如何组织的，也许/finance/revenue/ali-baba 可能对你来说最合适。

Subversion 可以移动目录，所以如果你的项目仓库变得失控了——可能你在根目录下放了几十个项目，导致一些奇怪的事情发生了——你可以轻易地把项目移出去。如第 82 页的 6.5 节“使用项目仓库 URL”中讨论的那样，用两个项目仓库 URL 去调用 svn move 做一个服务器端的重命名，立刻就能把目录移动好。你应该与开发人员协调好，确保他们都把改动签入，然后再执行移动操作，最后让每个人去运行 svn update 以获得新的目录结构。

8.3 多个项目仓库

把你所有的项目分布到不同的目录中去是很有道理的——开发者可以

很容易就找到他们应该改动的项目。把项目分布到不同的“项目仓库”中去也是有可能的。因为项目仓库在硬盘中是以特定目录中的一组文件的形式存放的，你可以在同一条服务器的不同目录中创建多个项目仓库，或者在互相独立的服务器上创建多个项目仓库。

如果你使用 `file://`或者 `svn+ssh://`这样的 URL 访问 Subversion 项目仓库，URL 的第一部分指定了项目仓库目录在服务器上的路径。你可以很容易地把它改成不同目录的项目仓库。

使用 `svnserve` 时，它的`--root` 选项指定了你项目仓库的“**虚拟根目录**”。比如说，如果你在虚拟根目录下创建了名字叫 `repos1` 和 `repos2` 的目录，每个目录中都有一个项目仓库，这些项目仓库的名字成为了 URL 的一部分。在本例中，你可以使用 `svn://myserver/repos1/..`来访问 `repos1`。

如果你是使用 Apache 来让 Subversion 项目仓库联网的，你可以给服务器上的每个项目仓库定义虚拟目录。在第 157 页的附录 A 中提到 Apache 的配置。

当然，把项目分布到不同的项目仓库中是额外的管理负担——你必须单独地备份每个项目仓库。用户可能需要更多信息才能找到他们想要的项目。额外的管理负担带来的好处是灵活性。如果你需要把一个项目仓库停机维护，[1]你可以做你想做的，而不会影响其他的项目仓库。如果某个项目长大到了不适合它现在存放的服务器，可能就须把项目仓库分成两个，这时可以添加第二服务器。

你不用在第一天就作出最终的决定。Subversion 提供了工具让你在项目仓库之间迁移数据，因此你可以在你知道了更多的需求之后改变主意。为了让事情尽可能地简单，我推荐你只使用一个项目仓库，直到你遇到了一个具体的问题，必须使用多个项目仓库才能解决，那时再去迁移也不迟。

---

[1] 这几乎是不大可能发生的，因为大部分 Subversion 维护，包括备份在内，都能在不停机的情况下完成。

第 9 章

使用标签和分支

Using Tags and Branches

Subversion 的日常使用是相当简单的：你先从项目仓库那里获取更新，然后编辑文件，在测试之后再把改动存回去。但是对于标签和分支，许多开发者都被它们难住了。可能他们之前所在的团队滥用了分支，导致项目仓库的结构就像一碗意大利面条，乱糟糟的，从而没法井井有条地开发。可能他们曾经待过的团队对于分支的合并总是一拖再拖，所以最终要去合并的时候就变成了一场解决冲突的噩梦。可能只是因为分支带来的灵活性太大了；有那么多选择，要知道该做什么很困难。

实际上，标签和分支可以（而且应该）用起来很简单。真正困难的是把他们用在合适的场合。在本章中我们展示两个相信分支应该被团队采用的场合：发布新版本和给开发者一个试验的空间。

在这两个场合之外，我们建议你在给项目仓库添加分支之前努力想清楚。过度的分支可以很快地让任何项目的项目仓库没法用。

在我们进入具体的实战指导之前，需要在整体上讨论一下标签和分支。

9.1 标签和分支

你的 Subversion 项目仓库能包含大量的信息。除了典型项目中的那些大量的源代码文件，Subversion 还储存了每个文件的所有版本。时间维度连同位置信息意味着复杂性的爆炸——我们怎么可能跟踪得了这么多？标签是给一组文件取的符号名称，每个都有一个特定的版本号。你可以把标签看作你项目仓库的切片，标记了其中的每一件东西，如图 9.1 所示。

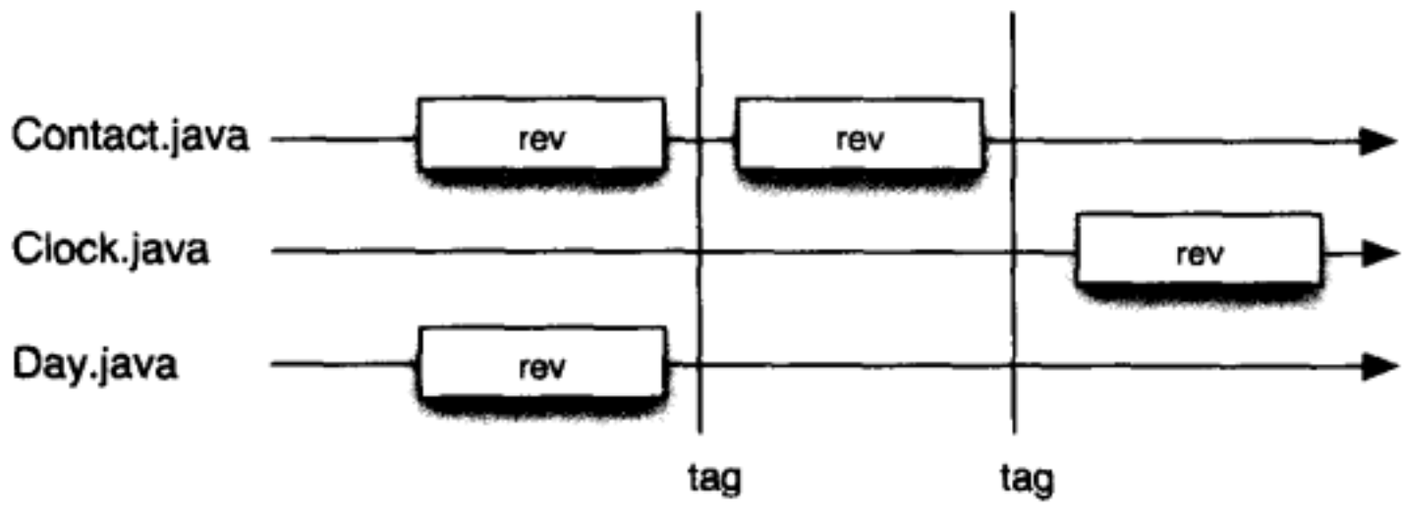


图 9.1 标签用作项目仓库的切片

标签对于跟踪项目生命周期中的重要事件是相当有用的。你不再要为了给客户发布一个版本记住需要使用版本 16 的 `Calendar.java`、版本 23 的 `Schedule.java`、以及版本 12 的 `contacts.dat`，相反你可以使用标签来帮你记住这些东西。因为 Subversion 的版本号也是项目仓库的切片，你可能认为我们可以只使用版本号或者日期就能签出所有的代码来构建一个发布版本。这可能是行得通的，但是标签可以是从**混合版本**的工作拷贝创建而来——你签出的文件并不是都对应于同一个项目仓库版本号。如果你想要挑选出项目组件的不同版本，在发布的时候打包到一起的话，就得这么做。

在 Subversion 中要打开一个标签，只须把你的代码（典型的做法是从主干中）拷贝到你项目的 `tags` 目录中。Subversion 处理拷贝的过程是非常有

\// Joe 问……
我如何让一个标签只读

标签只是你项目仓库特定版本的拷贝，所以没有什么可以阻挡人们把改动签入到标签目录。有时可以改动一个标签还是一件有用的事情，但是大部分时间最好还是把标签当作只读。

你可以把你的标签目录变成只读的（或者更正确地说是只可以创建的——新的标签应该是被允许的）通过使用第 174 页的 A.5 节“使用钩子脚本访问来做控制”中的项目仓库权限脚本中的一个。虽然，这经常是多此一举，因为开发者会在主干或者发布分支上做开发，而不会去动标签中代码。

效率的，立刻就拷贝完了，而且只需要很少的空间来存储它。你把代码拷贝到的那个目录是标签的符号名。拷贝起到引用标记的作用，把文件按标签创建时的原样存储在你的项目中。

Subversion 中的目录拷贝只是简单拷贝。习惯上来说，你绝对不会修改标签目录下储存的代码的，但是实际上没有什么会阻止你去这么做。如果你确实要改变标签目录中的内容，标签实际上就成分支了。Subversion 不会把它移动到你的分支目录或者做其他聪明的事情，但是标签包含的不再只是项目仓库的静态快照了。这在某些场合下是有用的——比如你可以设置一个名字叫“最新”的标签，总是包含你最新构建（并且测试了）的代码。

我们在第 19 页的 2.7 节“分支”第一次谈到了分支，那里我们讨论了如何使用它们来处理版本控制系统中的发布问题。分支代表了项目仓库历史的一个分叉；同一个文件可能有两个或更多的独立改动，分别存在于独立的分支之中。

要在 Subversion 中创建一个分支，你需要拷贝你的主干代码到你项目的 branches 目录下。新目录命名了分支，而且在分支被创建的最初只是 Subversion 中存储的文件的忠实拷贝。当你修改了分支中的文件后，Subversion 平行地记住了这个改动以及对原始主干的改动。Subversion 还记住了两个文件有共同的历史。

■ 实践中的标签和分支

标签和分支有大量可能的用途。然而，过度使用标签和分支最终都会把事情搞得极端混乱。所以为了把事情保持简单，建议你一开始只把他们用于四个用途：

发布分支

我们推荐把项目的每个发布版本放到一个单独的分支中。所使用的目录放在 branches 目录中，这个目录名就是分支的名字。

发布

发布分支会包含一个（可能是多个）分支：标记了项目发布的过程。发布标签标记了这些关键点。

Bug 修正

在发布过程中发现的 Bug 是在发布分支中修正的。如果有必要的话，修正的代码会被合并到主干，以及其他的发布分支中。在一次提交就能修正的 bug 的情况中，Subversion 的版本号就足以辨别改变了什么了，从而可以执行对应的合并操作了。对于更复杂的 bug，需要创建一个分支来修正这个 bug，并在修正完成了之后把改动合并到发布分支和主干当中去。使用标签来标记 bug 修正的开始和结束，以更容易地进行合并工作。

开发人员的试验

有时团队中的一部分人必须在项目的代码基础上做一些比较远的试验。在改动期间，代码是与系统的其余部分不兼容的，可能会让项目整体无法构建。开发者可以选择创建一个开发人员的试验分支，让他们在那儿去做他们的改动。

Thing to Name	Name Style	Examples
Release branch	RB-*rel*	RB-1.0 RB-1.0.1a
Releases	REL-*rel*	REL-1.0 REL-1.0.1a
Bug fix branches	BUG-*track*	BUG-3035 BUG-10871
Pre–bug fix	PRE-*track*	PRE-3035 PRE-10871
Post–bug fix	POST-*track*	POST-3035 POST-10871
Developer experiments	TRY-*initials-desc*	TRY-MGM-cache-pages TRY-MR-neo-persistence

图 9.2 标签和分支的命名规范

给整个团队制定标签和分支的命名规范是个好主意。图 9.2 中的表格显示了一个简单的模式；我们在本书中会使用这样的命名规范。在这个表格中，*rel* 指的是发布号，*track* 指的是 bug 跟踪号。

下面我们来看看实践中的分支和标签是如何使用的。从你项目生命周期中的常见事件入手（我们希望如此！）——创建一个发布分支从而我们可以发布一些代码。

9.2 创建发布分支

在软件被开发的过程中，你不时都须去发布一个版本。每次临近发布的时候，注意力就从添加新功能转移到清理与发布相关的细节上。虽然最初整个团队都可能参与这个过程，但是根据回报减少定律，划分出一个发布子团队，专注于为发布把代码准备好，会让工作更有效率一些。如果这个子团队

在主干上工作，团队其余人的工作就会被妨碍，要等到他们结束了才行。

在开发进行到这个阶段时，把要发布的代码移到它自己的分支中去。发布团队在分支上做开发时，项目的其余人员还能继续在主干上工作。当发布本身完成了，我们用发布号给发布分支打上标签（记住，标签只是发布分支在特定时刻的一个简单拷贝）。由发布团队在发布分支中做出的改动可以在之后被合并回主干去，如图 9.3 所示。

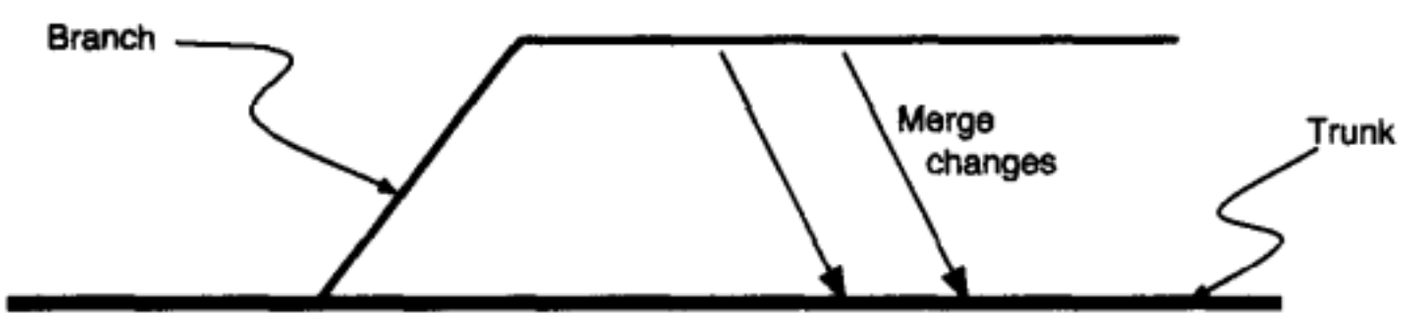


图 9.3 发布分支合并到主干

通过把项目的主干拷贝到 branches/下的一个新目录就能创建一个发布分支了。使用项目仓库 URL 来做这件事是最合适的，因为这样的话分支创建完全是在服务器端进行的，所以速度会快很多。你应该确保每个人都签入了他们本地的工作拷贝，并且为分支的创建做好了准备。[1]

在下面的例子中，我们为项目的 1.0 版本创建了一个分支。我们还须创建/sesame/branches 目录，因为它是我们创建的第一个分支：

```
work> svn mkdir -m "Creating branches directory"
                   svn://olio/sesame/branches
Committed revision 32.
```

[1] 使用项目仓库的 URL 来创建分支，可以让分支从项目仓库的任意版本开始。如果你创建分支时晚了一点，还是可以问问其他开发者，找出哪个版本是应该用来创建分支的，然后使用该版本来创建。

```
work> svn copy -m "Creating release branch for 1.0" \
                 svn://olio/sesame/trunk              \
                 svn://olio/sesame/branches/RB-1.0
Committed revision 33.
```

创建分支目录和创建实际的分支都须提交消息，因为它们都改动了项目仓库。

此时，我们所做的只是创建了发布分支。任何由开发者签出的工作拷贝仍然指向主干。为了使用发布分支，须把它签出到新的工作拷贝中去。

9.3 在发布分支上开发

要访问发布分支，你需要把项目从它的分支目录而不是其主干目录签出。你能够签出到另外一个目录，或者把现有的工作拷贝“转向”为指向分支。我们推荐前者；它更清晰而且同时简化了在两个分支上做开发的工作。`svn switch` 命令对于在工作拷贝中组装不同的代码分支会有用处，所以我们两种方式都加以讨论

switch

■ 签出一个发布分支

如果你像我们一样，有大量的硬盘空间并且宁愿浪费那么一点也不愿意去记住一个特定的工作拷贝是指向了哪个分支。我们倾向于给每个活跃的开发分支保持一个签出的工作拷贝，这么做只是为了让事情更简单一些。

切换回你的 `work` 目录，然后从分支目录签出，并改写默认的目录名，最终代码会被签出到 `rb 1.0` 目录下。当你签出分支时，你签出的是分支的最新代码；这与在主干中签出代码返回文件的最新开发拷贝是一样的：

```
work> svn co svn://olio/sesame/branches/RB-1.0 rb1.0
A rb1.0/Month.txt
A rb1.0/Number.txt
A rb1.0/common
A rb1.0/common/Log.java
A rb1.0/common/Clock.java
```

```
A rb1.0/Day.txt
A rb1.0/contacts
A rb1.0/contacts/Contacts.java
Checked out revision 33.
```

如果我们现在在这个签出的发布目录中编辑文件，并且提交改动，Subversion 会把改动添加到分支中，而不是主干中。现在可以接着为准备发布去改善文件了。

■ 把工作拷贝转向到发布分支

svn switch 命令改变工作拷贝的全部或者部分，让它指向不同的分支。因为大部分分支只包含很少一些与主干不一样的东西，Subversion 可以以极端有效率的方式执行这个操作，只把改动过的文件传输到客户端。把工作拷贝转向到不同的分支比起从那个分支签出一个新的工作拷贝来要快得多。

要把指向 Sesame 主干的工作拷贝转向到 1.0 发布分支，运行下面这条 svn switch 命令：

```
work> cd sesame
sesame> svn switch svn://olio/sesame/branches/RB-1.0
U  common/Clock.java
U  contacts/Contacts.java
Updated to revision 36.
```

Subversion 更新 Sesame 工作拷贝中的文件让其包含发布分支最新的文件，在本例中，更新的是两个 Java 文件，因为它们是对发布分支做的 bug 修正。

svn switch 命令还接受--revision 参数，来指定你想要转换哪个版本分支。在默认情况下，Subversion 转向到分支的最新版本（即 HEAD 版本）。

你可以把工作拷贝转换回主干，像这样：

```
sesame> svn switch svn://olio/sesame/trunk
U  common/Clock.java
U  contacts/Contacts.java
Updated to revision 36.
```

Subversion 还能转换一个子目录，甚至单独一个文件到不同的分支。下一节中将用它来组装不同的版本，来包含客户指定要做的 bug 修正。

9.4 发布

在所有的调整结束，验收测试做完了之后，团队决定发布。最重要的是确保我们在正确的分支上给正确的文件组合打上了标签，从而我们确切地知道在发布的东西中都有什么。

创建一个发布标签最简单的方式是把分支拷贝到 tags 下的一个新目录。这样会给发布分支的最新代码打上标签。

有时你可能想要给不处于同一版本的文件或者目录打标签。一般而言，需要给不是分支中最新的代码打标签往往意味着某些地方做得不对，所以我们不推荐养成这么去做事的习惯。Subversion 可以给任何工作拷贝的状态打标签，拷贝一个混合版本到标签中，从那里你就可以发布了。

这两种方法在起初会有一些令人迷惑，所以从最简单的做法开始。一旦对于发布分支的最新代码满意了，把它拷贝到 tags 下的一个新目录中：

```
work> svn mkdir -m "Creating tags directory" \
          svn://olio/sesame/tags
Committed revision 34.
work> svn copy -m "Tag release 1.0.0"         \
          svn://olio/sesame/branches/RB-1.0 \
          svn://olio/sesame/tags/REL-1.0.0
Committed revision 35.
```

之前的 svn copy 拷贝了最新的代码，在本例中是版本 34，从发布分支到新标签 REL-1.0.0。

有时，你需要给不是分支最新的代码打标签。假设发布 1.0.0 是几个月前的事情了，而且团队已经成功地发布了一些小的 bug 修正，把版本号升到了 1.0.4。一个重要的客户需要给 1.0.0 版修正一个 bug，但是他又不想等着 1.0.5 出来再修正它。bug 是很简单的错误，所以我们决定在 1.0.0 的代码中修正它，并且发布给客户。[2]

[2] 这是相当不好的做法，但是如果你的客户真的不想升级，这也可能是唯一的解决方案。

我们签出一个工作拷贝，其中有 1.0.0 的所有东西，原样不动。然后为我们的客户更新一些文件。在这个例子中，Clock.java 包含了我们想要修正的 bug：

```
work> svn checkout svn://olio/sesame/tags/REL-1.0.0 \
          client-fix
A client-fix/Month.txt
A client-fix/Number.txt
A client-fix/common
A client-fix/common/Log.java
A client-fix/common/Clock.java
A client-fix/Day.txt
A client-fix/contacts
A client-fix/contacts/Contacts.java
Checked out revision 37.
```

接下来使用 svn switch 来改变 common 目录的指向。我们能从发布分支中得到 bug 修正代码，因为在 1.0.0 标签创建之后修正了它，所以我们转向到 RB-1.0：

```
work> cd client-fix
client-fix> svn switch                                          \
            svn://olio/sesame/branches/RB-1.0/common \
            common
U common/Clock.java
Updated to revision 37.
```

现在你的工作拷贝包含了客户想要的东西了——与 1.0.0 发布时一样的代码，以及我们在发布分支中对一个关键 bug 的修正代码。

在运行了测试，验证了我们的工作拷贝确实修正了那个问题之后，可以创建一个新的标签了。我们使用 svn copy 来把 client-fix 工作目录拷贝到新的标签目录 REL-1.0.0-clientfix 中：

```
client-fix> cd ..
work> svn copy -m "Tagging client's 1.0.0 fix" client-fix \
          svn://olio/sesame/tags/REL-1.0.0-clientfix
Committed revision 37.
```

开发者可以获得用来构建一个特殊发布的代码，使用 svn checkout 和标签的 URL：

```
work> svn co svn://olio/sesame/tags/REL-1.0.0
A  REL-1.0.0/Month.txt
A  REL-1.0.0/Number.txt
A  REL-1.0.0/common
A  REL-1.0.0/common/Log.java
A  REL-1.0.0/common/Clock.java
A  REL-1.0.0/Day.txt
A  REL-1.0.0/contacts
A  REL-1.0.0/contacts/Contacts.java
Checked out revision 37.
```

9.5 在发布分支中修正 bug

bug 随时都会被发现。困难的是如何井井有条地处理它们。在发布分支中，这意味着须需要跟踪我们修正 bug 所做的改动，然后确保把这些修正代码应用到其他可能包含相同问题的分支之中。最后一点尤其重要。由于分支的本质决定了，它们包含着重复的代码。这意味着如果你在一个分支的源代码中找到了一个 bug，总是有可能在另外一个分支中找到相同的 bug（毕竟，原始的源代码都是一样的，bug 也在其中）。如果分支是发布分支，我们要能够把我们的修正代码应用到主干。可能还要把它应用到其他发布分支中（如果他们也包含有 bug 的代码的话）去。

没有版本控制的话，这是一个高难度的问题。有了版本控制，我们可以更好的管理做事的流程。在修正 bug 的时候让版本控制系统来跟踪源代码的改动，然后把改动合并到其他受影响的分支中去。

使用 Subversion，对 bug 修正跟踪到什么程度取决于 bug 有多“难”修复。如果它是一个小 bug，修正代码可能不过是改动几个文件中的几行代码。对于更严重的缺陷，修正可能包括改动相当数目的代码，而且要添加和移除一些文件，这可能是一个团队的行为，需要一个以上的开发者的参与。

Subversion 使用版本号来跟踪改动，像我们在第 42 页的 3.6 节“更新项目仓库”中看到的那样。如果你可以在一次提交中修正 bug，只要记住那个版本号就足以用来把改动拷贝到其他分支了。如果 bug 更复杂一些，须多次提交才能修正（或者经过数次失败的尝试才修正好），你可能须创建一个分支来跟踪这次修正过程。

■ 修正简单的 bug

假定我们想要修正的是相对简单的问题，修正这样的问题只须改动几个文件。这个过程描述在下面的列表中：

1. 签出包含 bug 的代码到本地工作拷贝。

2. 编写测试重现 bug，修正代码以让新的测试通过，并且验证构建过程还能通过。

3. 把你的改动提交到项目仓库中，并且记住新的版本号。记住这个版本号的好办法是把它添加到你的 bug 跟踪系统中，从而每个人都能在以后找到它。

4. 使用新的版本号把改动合并到其他受影响的分支中（可能还包含主干）。

作为一个例子，让我们在发布分支中修正 bug 3065。首先，去发布分支的工作拷贝中，修正 bug，并签入：

```
rb1.0> # .. edit contacts/Contacts.java and fix the bug .. #
rb1.0> svn commit -m "Fix bug 3065 (address formatting)"
Sending        contacts/Contacts.java
Transmitting file data .
Committed revision 38.
```

Subversion 告诉我们，修正代码被提交到版本 38 了。为了把修正代码合并到主干中，我们去主干工作拷贝中，要求 Subversion 合并版本 38。更具体说，我们要求把版本 37 和 38 之间的差异合并到主干工作拷贝中：

```
rb1.0> cd ../sesame
sesame> svn update
At revision 38.
sesame> svn merge -r37:38 svn://olio/sesame/branches/RB-1.0
U contacts/Contacts.java
sesame> svn commit -m "Merge r38 (fix bug 3065)"
Sending        contacts/Contacts.java
Transmitting file data .
Committed revision 39.
```

■ 复杂的 bug

如果是处理复杂的 bug，可能需要数个开发者用几天的时间才能修正好。就不能用 Subversion 的版本号这种方式来处理了。廉价拷贝再一次拯救了大家——我们会创建一个分支，bug 修正可以在那里进行并且使用标签来标记我们修正过程的开始和结束。这些标签可以帮助我们把修正代码合并到其他分支中。这个过程走一遍下来大概是这样：

1. 把包含 bug 的代码分支到新的 bug 修正分支中。

2. 给新分支打个标签，标记 bug 修正开始。
3. 编写测试重现 bug，修正代码以让新的测试通过，验证构建过程还能通过。
4. 提交你的改动到项目仓库。如果需要尝试好几次才能修正 bug，也没什么好担心的。
5. 一旦你对修正代码满意了，再给分支打一个标签，标记 bug 修正结束。
6. 使用两个标签来把修正代码合并到所有其他受影响的分支。

在给 bug 修正创建一个分支时，我们使用命名习惯 BUG-*track*，其中 *track* 是 bug 跟踪号。我们使用名字叫 PRE-*track* 和 POST-*track* 的标签来标记修正 bug 活动的开始和结束：

```
work> svn copy -m "create bugfix branch"          \
          svn://olio/sesame/branches/RB-1.0       \
          svn://olio/sesame/branches/BUG-10512
Committed revision 40.
work> svn copy -m "tag bugfix start"              \
          svn://olio/sesame/branches/BUG-10512    \
          svn://olio/sesame/tags/PRE-10512
Committed revision 41.
```

我们已经使用标签 PRE-10512 标记了 bug 修正的开始，接着就可以在分支 BUG-10512 中做实际的 bug 修正工作。签出一个包含了分支代码的新工作拷贝，修正这个 bug：

```
work> svn checkout svn://olio/sesame/branches/BUG-10512
A  BUG-10512/Month.txt
A  BUG-10512/Number.txt
A  BUG-10512/common
A  BUG-10512/common/Log.java
A  BUG-10512/common/Clock.java
A  BUG-10512/Day.txt
A  BUG-10512/contacts
A  BUG-10512/contacts/Contacts.java
Checked out revision 41.
work> cd BUG-10512
BUG-10512> # ..Fix bug, possibly adding and removing files .. #
BUG-10512> svn commit -m "Fixing bug 10512"
Adding          Year.txt
Sending         common/Log.java
Transmitting file data ..
Committed revision 42.
BUG-10512> # .. Bug wasn't fixed, ask Bob to help out too .. #
BUG-10512> svn commit -m "Still fixing bug 10512"
Sending         Number.txt
Transmitting file data .
Committed revision 43.
```

至此，我们已经修正了 bug。它让我们试了好几次，甚至可能还要让同事去签出 BUG-10512 分支的代码，让他为我们再看看。现在应该给修正 bug 的分支打上标签，从而可以识别出 bug 修正的结束：

```
BUG-10512> cd ..
work> svn copy -m "tag bugfix finish"          \
          svn://olio/sesame/branches/BUG-10512 \
          svn://olio/sesame/tags/POST-10512
Committed revision 44.
```

现在把 bug 修正代码合并到发布分支，那里是我们一开始想要去修正 bug 的地方。在合并之后，运行测试以确保没有东西被破坏，然后签入：

```
work> cd rb1.0
rb1.0> svn update
At revision 44.
rb1.0> svn merge svn://olio/sesame/tags/PRE-10512 \
                 svn://olio/sesame/tags/POST-10512
U Number.txt
U common/Log.java
A Year.txt
rb1.0> # ... run tests ... #
rb1.0> svn commit -m "Merged fix for bug 10512"
Sending        Number.txt
Adding         Year.txt
Sending        common/Log.java
Transmitting file data ..
Committed revision 45.
```

同样，svn merge 命令可以用来把 bug 修正代码拉到其他分支以及主干中。只要切换到对应的工作拷贝，确保它们是更新过了的，并且使用了同样的合并命令去获得修正代码。

大部分情况下，使用版本号这种简单的方式来修正 bug 是行之有效的，所以只要可以的话就用这种方式。有些 bug 跟踪软件能够跟踪版本号，但是如果你使用的软件不支持的话你只要把版本号放在 bug 的注释栏就可以了。

9.6 开发者的试验分支

有时开发者须给项目做出牵涉面很广的改动（比如说，改变持久层以引入一种新的安全机制）。这种事情至少要写上好几天，并且（不幸的是）它们不能逐步地引入：它们实在是影响了太多的代码了。这些改动一般是在应

用程序的底层，并且一般对系统的其余部分有着深远的影响。

如果一个开发者想要对源代码做一个大范围的改动，他们可以在本地做。然而，这可能有几点不好的地方。首先，开发者在开发的时候失去了版本控制的好处。其次，他们不能回退自己的工作了，失去了版本历史，等等。他们还没有把工作放在中心的项目仓库中，因此可能就会忘记做备份。

如果多个开发者在开发一个做大范围改动的东西，那么会有更大的问题；他们要能够共享改动，并且工作在同一个（实验性质的）代码库中。

解决方案是把试验代码放在版本控制系统中的分支中。在做那方面开发的开发人员在他们的工作拷贝中使用那个分支。当他们做完工作之后，可以做出决定是不是要把工作集成到主干中。如果试验失败了，可以废弃这个分支。否则他们只要把分支的改动合并到主干中就可以了。无论他们的决定是什么，将来的开发仍旧在主干中进行，分支成了历史。

创建开发者分支和创建发布分支是一样的。把主干拷贝到实验性分支目录，与其他的分支储存在一起就行了：[3]

```
work> svn copy -m "new hibernate persistence spike" \
                svn://olio/sesame/trunk                \
                svn://olio/sesame/branches/TRY-MGM-hbn-spike
Committed revision 45.
```

为了开始使用分支，你要么把它签出到新的工作拷贝中要么把现有的工作拷贝转向到新的分支。

[3] 你可能不想把实验性分支和你发布分支目录混杂在一起，Subversion 完全接受你把分支放在任意你喜欢的位置。只要你记得住/branches/cb/fluffy 包含了新的持久化框架……

9.7 开发实验性代码

如果你已经签出了一个工作拷贝，可以使用 svn switch 把它转向新的试验分支。这儿我们要把 sesame 工作拷贝转向：

```
work> cd sesame
sesame> svn switch svn://olio/sesame/branches/TRY-MGM-hbn
        -spike
At revision 45.
```

要把 sesame 工作拷贝转回主干来，可再次使用 svn switch：

```
sesame> svn switch svn://olio/sesame/trunk
At revision 45.
```

如果不想重用工作拷贝，你可以把分支签出到新的目录。这是我们更喜欢的方式，因为它让你更清楚地知道自己在开发什么：

```
work> svn co svn://olio/sesame/branches/TRY-MGM-hbn-spike
      hbn-spike
A  hbn-spike/Month.txt
A  hbn-spike/Number.txt
A  hbn-spike/common
A  hbn-spike/common/Log.java
A  hbn-spike/common/Clock.java
A  hbn-spike/Day.txt
A  hbn-spike/Year.txt
A  hbn-spike/contacts
A  hbn-spike/contacts/Contacts.java
Checked out revision 45.
```

9.8 合并试验分支

一旦你对在试验分支中做过的改动满意了，要把它们合并回主干去。要达到这个目的，首先要确保所有的开发者已经签入了他们的改动，并且你有一个指向主干并且更新了的工作拷贝（这也就是我们建议把试验分支签出到不同目录的原因）。

我们需要让 Subversion 把试验分支从它被创建时的状态到最新的状态之间所做过的所有改动合并到主干中。为此，需要知道分支是啥时候创建的。幸运的是，svn log 有一个--stop-on-copy 选项正能告诉我们这事：

```
work> svn log --stop-on-copy \
          svn://olio/sesame/branches/TRY-MGM-hbn-spike
```

```
------------------------------------------------------------------------
r47 | mike | 2004-11-12 13:47:13 -0700 (Fri, 12 Nov 2004)
Added hibernate utils
------------------------------------------------------------------------
r46 | mike | 2004-11-12 13:46:27 -0700 (Fri, 12 Nov 2004)
Made Contacts a hibernate mapped class
------------------------------------------------------------------------
r45 | mike | 2004-11-12 12:55:21 -0700 (Fri, 12 Nov 2004)
new hibernate persistence spike
------------------------------------------------------------------------
```

它告诉我们 TRY-MGM-hbn-spike 分支在版本 45 被创建（Subversion 还告诉我们创建这个分支的时候，但是可能在我们想要合并的时候已经忘记这些了）。现在我们可以把所有在版本 45 和 HEAD 之间的改动合并到我们主干的工作拷贝中了：

```
work> cd sesame
sesame> svn update
At revision 47.
sesame> svn merge -r 45:HEAD \
            svn://olio/sesame/branches/TRY-MGM-hbn-spike
A  common/HibernateHelper.java
A  contacts/Contacts.hbm.xml
U  contacts/Contacts.java
```

至此我们在合并中解决了所有的冲突，运行单元测试确保一切正常，然后签入：

```
sesame> # .. run unit tests to make sure everything's ok .. #
sesame> svn commit -m "Merged TRY-MGM-hbn-spike to the trunk"
Adding         common/HibernateHelper.java
Adding         contacts/Contacts.hbm.xml
Sending        contacts/Contacts.java
Transmitting file data .
Committed revision 48.
```

在本章中用到的技术对应数个 SCM 模式。使用发布分支对应于“release line”和“release-prepare codeline”模式。实验性开发分支对应于“task branch”模式。分支通常与“代码行规则”（哪怕规则是很不正式的）相关，可以帮助开发者明白他们应该如何处理每个分支上的代码。

第 10 章

创建项目

Creating a Project

很难说得清楚“项目”是什么。一个人一周就可以写出来的 web 表单可以称作项目，一百个人干上数年的也能称作项目。但是大部分的项目有一些共同的特征：

- 每个项目都有名字。这可能听起来就像没说一样，但是我们在想要识别单独的个体时，总是倾向于给东西取上名字。名字不一定是对外的商标，由市场部门批准并且要在大城市去做市场调查。项目名只是在你的组织内部使用的。
- 每个项目都是内聚的；项目的组件一起达成一些业务目标。
- 项目内的组件倾向于作为单元来维护；你要把项目的一个版本作为整体来发布。
- 项目中的参与人员遵守同一套工程标准和规范，并且使用共同的架构。

在把项目放入版本控制系统时考虑一下这个列表是很重要的，因为经常会发现有时很难在项目之间划清边界。把项目结构搞错了是使得版本控制混乱的一个主要来源，并且随着时间的推移还可能导致大量的工作浪费。Subversion 确实是让项目开在始了之后把东西移来移去时直接了许多，但是

这也需要与使用项目仓库的所有人都协调沟通好。

Subversion 使用目录来组织所有的东西，因此项目将对应于你项目仓库中的一个目录的位置。子项目可能对应于子目录，等等。这种做法给了你灵活性，让你能按照你认为最适合的项目目录结构来组织内容，但是即便如此，要知道从哪里开始还是有点困难的。

因此，在你的项目仓库中创建项目之前，花一些时间做计划。比如，你的项目是不是要实现一个框架，整个公司都要在以后的开发中使用？如果是这样，那么这个框架可能要独立出来作为一个项目，供当前的项目和其他未来的项目共同使用它。你的项目是不是要开发多个独立的组件？可能每个都要成为一个单独的项目。或者你的项目是不是要给一些已有的代码做扩展？那样的话它应该成为原来项目的子项目。

10.1 创建初始项目

在 Subversion 项目仓库中基本上有三种创建目录（也就是项目）的方式：

- 把已有的源代码导入到项目仓库的目录中。
- 使用`svn mkdir`手工创建目录，直到你弄出了一个想要的项目结构为止。
- 转换已有的源代码仓库。有Subversion工具可以转换CVS，RCS，Visual SourceSafe以及Perforce的项目仓库。

从其他的版本控制系统做转换是一个很大的话题，在第 181 页的附录 B 中有详细的讲述。除此之外，我们还有两个选择：导入和手工创建目录。

■ 导入到 Subversion

如果你有现有的源代码文件（哪怕它只是项目的 README 文件），也可以使用 `svn import` 命令把那些文件拿到你的项目仓库中来。在下面的例子中，

将假设在开发一个 Wibble 项目（Wickedly Integrated Business-to- Business Lease Exchange）。

你需要一个要导入的目录树包含着所有的文件（而且只包含有你想要导入的文件；在进行下一步之前，确保已经清除了各种备份文件以及其他的垃圾）。确信你位于目录树的最上层（在我们的例子中，是在目录 wibble 中），然后输入 svn import 命令：

```
wibble> svn import -m "Wibble initial import" \
          svn://olio/wibble/trunk
Adding         wibble.build
Adding         src
Adding         src/Wibble.cs
Adding         src/WibbleTest.cs
Adding         README
Committed revision 49.
```

它将告诉 Subversion 把当前目录的内容导入，存储到项目的 /wibble/trunk。Subversion 在导入的过程中根据需要自动创建了父目录，但是你可能还想要创建项目的 tags 和 branches 目录：

```
wibble> svn mkdir -m "Create tags directory"     \
          svn://olio/wibble/tags
Committed revision 50.
wibble> svn mkdir -m "Create branches directory" \
          svn://olio/wibble/branches
Committed revision 51.
```

项目现在被签入了。你应该使用 svn checkout 签出，如果一切正常话，你可以删除原来导入的目录树。

■ 手工创建目录

如果你的项目还没有文件，一种简单的开始的办法是创建一个空的目录结构，然后逐步添加文件。

可以使用 svn mkdir 命令在项目仓库中创建目录。对于 Java 项目，你可能想要像图 10.1 中那样的目录结构。svn mkdir 能帮助你用一条命令创建多个目录。

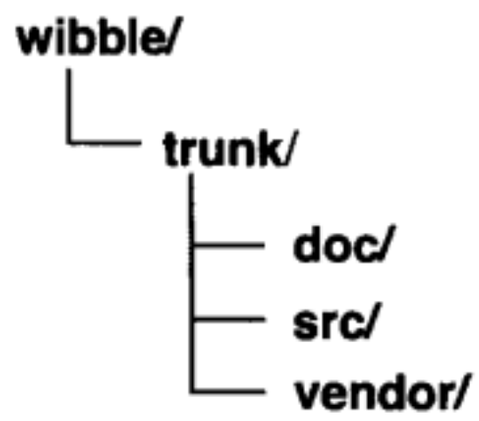


图 10.1 Wibble 项目结构

```
work> svn mkdir -m "Creating initial structure" \
          svn://olio/wibble                      \
          svn://olio/wibble/trunk                \
          svn://olio/wibble/trunk/src            \
          svn://olio/wibble/trunk/doc            \
          svn://olio/wibble/trunk/vendor
Committed revision 54.
```

Wibble 项目现在已经可以签出到本地拷贝来了。你能够像你日常开发时所做的那样，添加源代码，文档和库，用 `svn add` 命令添加文件。

10.2 项目内部的结构

你的公司可能已经制定好了标准，专门指导如何去组织源代码和目录。比如说，如果你使用 Java 开发，可能需要使用 Jakarta 习惯来排列目录。[1]如果现在还没有确定一个标准，下面有一些基本的建议。

[1] http://jakarta.apache.org/site/dirlayout.html

■ 顶层文件

这些是你在每个项目的顶层会找到的典型文件：

README

数年之后，现在还红红火火的项目可能会变得暗淡无光，虽然这事看起来难以置信，但是要“真正”回忆起 Wibble 项目是干啥的还真不容易。所以创建一个叫 RAEDME 的文件放在项目目录的最顶层。写一小段文件描述这个项目：它解决的业务问题，使用的基本技术，等等。这并不意味着要完整的描述，只要能在很长一段时间之后再来看它时，能够激活那些长期休眠的神经元，起到帮助回忆的作用。

BUILDING

创建另外名字叫 BUILDING 的顶层文件，包含了简单的提示给日后的代码考古者看，他们要做一项谁都不愿做的工作：从项目的源代码重新构建。因为你会把构建自动化，所以这篇文档应该很短；图 10.2 展示了一个例子。

GLOSSARY

再创建一个名字叫 GLOSSARY 的顶层文件。把所有与项目相关的术语记录在这个文件中当成一个习惯。这不仅仅有益于将来的开发人员使之更容易明白什么是“wibble_channel”，而且还可以给项目团队起指南的作用，指导如何命名类，方法和变量。

■ 顶层目录

大部分项目至少有以下的顶层目录：

doc/

把所有的项目文档签入到 doc 以及其子目录中。不要忘记把描述了决策的备忘录和邮件也放进去。常见的做法是在 doc 下有一些目录包含不同的文档类型或者对应于项目的不同阶段。

```
Prerequisites:
  * Oracle 9.6i (perhaps later versions but
    that configuration's not tested)
  * GCC 2.96
Building:
  ./configure [--with-oracle=<dir>]
  make
  make test
  make install
More info:
  docs/building.html
```

图 10.2 简单的 BUILDING 文件

如果你的项目需要外部的文档（比如说，一个算法的描述或者一个第三方文件的格式），考虑把这些拷贝过来存放在 doc 目录树下（当然是版权允许的情况下）。这让以后的维护者在外部的网站挂掉了的情况下也能轻松维护。如果你不能拷贝这些材料到你的项目之中，则可在 doc 中创建一个叫 BIBLIOGRAPHY 的文件，并且在其中添加一些链接和简略的描述。

data/

许多项目都带一些数据（比如，在数据库中填充查找表所需的信息）。要把这些数据放在一个一定的位置（如果没有别的理由的话，就这么去做，比如说某人某天急着要找出为什么我们在关岛要收取百分之一百二十七的销售税，我们就能让它去这里找答案了）。

db/

如果你的项目使用数据库，把所有表结构相关的东西都放在这里。努力养成不要在线修改表结构的习惯。让你的数据库管理员给每次更新编写 SQL 脚本——既更新表结构“同时”也迁移数据。把这些放在项目仓库中，你就可以把任何版本的数据库迁移到其他的版本上去。

src/

项目的源代码应该放在这个目录之下。你可能需要子目录来隔开不同种类的源代码，比如 src/java 和 src/eiffel。

util/

这个目录装着各种与项目相关的实用程序，工具和脚本。有些团队把这个目录命名为 `tools`。

vendor/

如果你的项目使用了第三方库或者头文件，你想要和你自己的代码一起存档，可以把它放在顶层的 `vendor` 目录中。

vendorsrc/

有时一个项目会从第三方导入或者包含代码（比如，如果它使用了一个开源的库，并且须确保在程序的生命周期内都能一直访问到特定版本的源代码）。你把库的二进制文件（可能还有头文件）放在 `vendor` 目录，但是你还想要保留构建这些库的源代码。可以把这些源代码存放在 `vendorsrc` 目录下。我们在第 147 页的 11 章“第三方代码”中对此有更多要说的。

在图 10.3 中有一个 Wibble 项目可能的文件布局方案。在这个项目中，我们有自己的源代码（分成客户端和服务器两部分）以及一些导入的开源代码（JUnit 和 Spring 框架）。

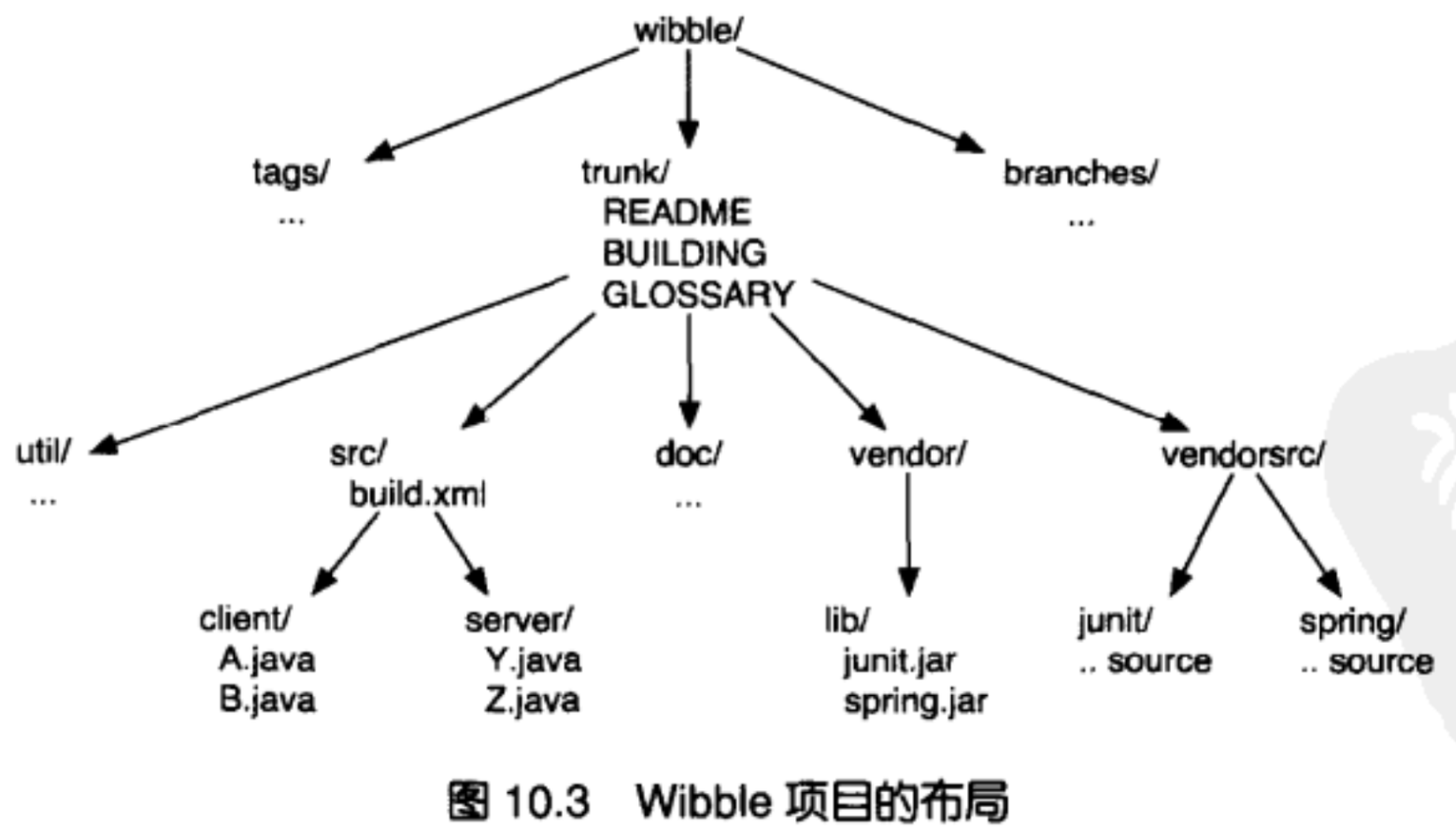


图 10.3 Wibble 项目的布局

除此之外，许多项目还有一套标准的目录用于构建或者发布项目。这些目录不包含应该存储在项目仓库中的文件（因为它们的内容是动态产生的），但是一些团队仍然觉得要是这些目录能出现在每个开发者的工作拷贝中会很方便。要达到这个目的，可以添加一些空的目录到项目仓库之中；他们会在开发者签出的时候出现在工作拷贝中。

另外一个同样有效的做法是不把这些目录存储在 Subversion 之中。相反，根据需要来构建脚本，然后在用完之后再把它清理干净。如果你采用这种方式，你可以把目录的名字添加到项目顶层目录的 `svn:ignore` 属性之中，免得 Subversion 整屏的问号把你淹死。

你还会想要把测试代码放在其他地方，但是对此各方意见有分歧。有些团队喜欢把它放在与代码树平行的目录中；其他人喜欢把测试放在被测代码的子目录中。某种程度上，“正确”的答案取决于你使用的编程语言。例如，Java 包命名规则意味着你如果想要测试 protected 方法，就需要构建平行的树（或者把你的测试放在被测源代码的同一个目录中）。我们的兄弟书“单元测试之道”[HT03]，[HT04]中对此有更详细的描述。

对于如何构造项目的目录结构没有绝对的金科玉律。然而，在不同项目之间遵循同样的习惯会帮助以后的人们，让他们在项目之间移动更容易，不会有“完全迷失方向”的感觉。

10.3 在项目之间共享代码

很少有存在于真空中的项目，相反项目总是会依赖组织内的其他项目的。一旦一些项目开始成熟，你时常会发现他们有一些共通的功能可以在项目之间重用。在大型企业中，经常可以看见一个团队专门开发可重用的框架和库。

在 Subversion 中“一切皆目录”意味着你的公共代码必须放在项目仓库的共享目录中。主要有两种方式可以达到这个目的：

- 把你的所有项目都存在一个“Uber-项目”中并且使用构建脚本来管理项目之间的依赖关系。
- 在你构建脚本之前，可使用`svn:externals`来把每个项目的依赖关系纳入你的工作拷贝中。

两种方式都行得通，但是对于项目的组织和分支来说使用 `svn:externals` 更加复杂。

■ 在 Uber-项目中共享代码

使用 Uber-项目时的项目仓库目录结构显示在图 10.4 中。

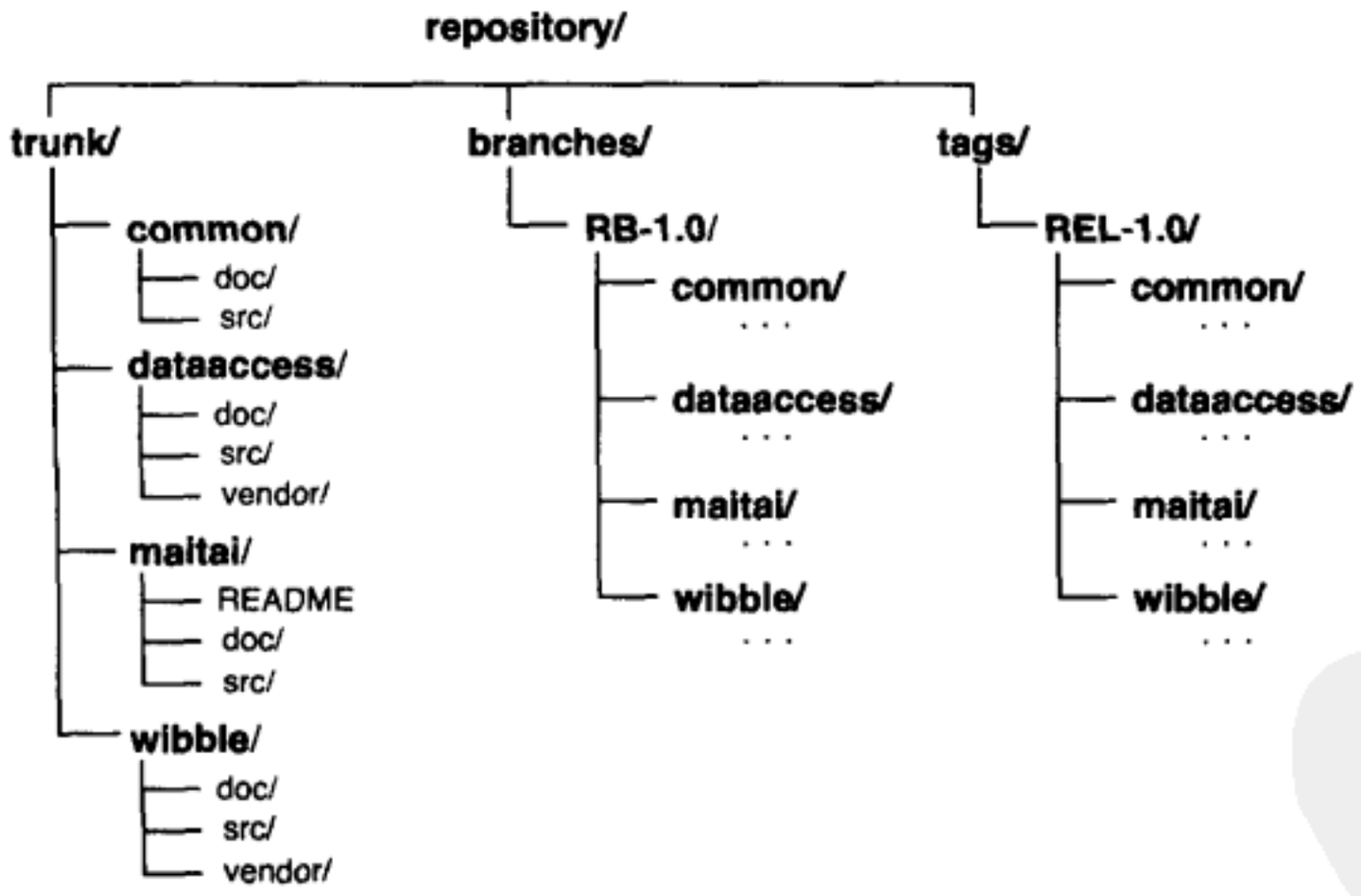


图 10.4 Uber-项目的项目仓库结构

`trunk/`目录下直接的目录对应于一个个的共享项目，在本例中是 `common` 和 `dataaccess`。目录 `maitai` 和 `wibble` 储存着 MaiTai 和 Wibble 项目的实际代码。

开发者把这个 Uber-项目签出到一个工作拷贝中，包含了所有的项目。把这个工作拷贝叫什么需要创意和灵感，想不出好的来时，就叫 uber-project 好了：

```
work> svn checkout svn://olio/trunk/ uber-project
A  uber-project/wibble
A  uber-project/wibble/doc
A  uber-project/wibble/doc/UserRequirements.doc
A  uber-project/wibble/src
A  uber-project/wibble/src/WibbleTest.java
     :       :       :
A  uber-project/dataaccess/lib/neo-1.3.0.dll
A  uber-project/dataaccess/src
A  uber-project/dataaccess/src/DataMapper.cs
A  uber-project/dataaccess/README
Checked out revision 17.
```

开发人员去构建 MaiTai 项目的时候，他们（几乎是立刻）期望脚本构建的是 MaiTai 依赖的项目。在本例中首先构建的是数据访问项目。一旦所有的依赖都构建完了——并且假设数据访问项目产生了一个库作为它构建的一部分——MaiTai 项目就使用这个库，开心地构建自己去了。

这个策略有几项缺点。首先，开发者必须签出你项目仓库中的所有项目的所有代码。这可能是不必要的，如果你的项目仓库很大，或者一些敏感代码是需要严格访问权限控制的。其次，分支选项受到了限制——你必须要把所有的代码同时分支，而不是按项目单独来做。

■ 用外部依赖共享代码

一个特殊的 Subversion 目录的属性，svn:externals 让你把另外一个项目仓库的内容包含在你工作拷贝中。在第 69 页的 6.4 节“属性”中对于什么是 Subversion 的属性，如何操作它们有完整的论述。

svn:externals 属性设置在目录这个级别，并且指定一个项目仓库 URL 的列表。在签出的时候会包含这些 URL。你可以在外部定义中使用任意的 Subversion 项目仓库——客户端会为你做好签出工作。这意味着可以从不是你直接控制的 Subversion 项目仓库那里包含代码，比如说，放在互联网上开源项目。

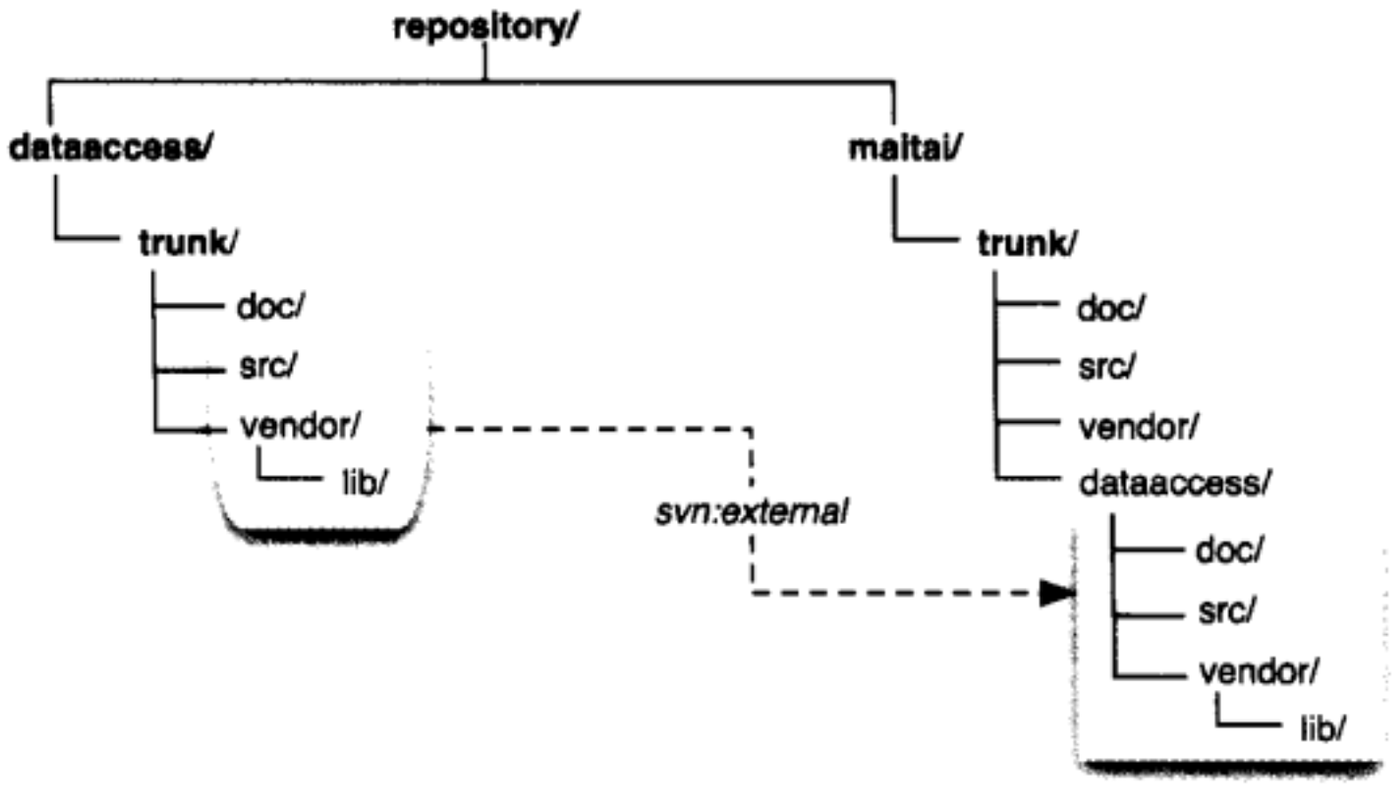


图 10.5 使用外部依赖的项目仓库布局

图 10.5 和图 10.6 展示了一个使用外部依赖来把共享代码链接到指定项目的项目仓库的样子。对此有很多值得一说的，以下一道来。

首先来看 MaiTai 项目。它储存在/maitai/trunk 中(显示在图 10.5 中)。它依赖数据访问项目，位于/dataaccess/trunk。在方框中的文件名是要以外部依赖的方式引入到 MaiTai 树之中的文件。为了设置这些依赖，我们签出 MaiTai 项目并且设置 svn:externals 属性：

```
work> svn checkout svn://olio/maitai/trunk maitai
A maitai/lib
A maitai/src
Checked out revision 19.
work> svn propset svn:externals                          \
          "dataaccess svn://olio/dataaccess/trunk" \
          maitai
property 'svn:externals' set on 'maitai'
```

这儿我们设置 svn:externals 属性，只引入了一个外部依赖。你可以使用 svn propedit 打开一个编辑器，如果你有多项依赖需要添加的话，其中每个都应该列在单独的一行中。

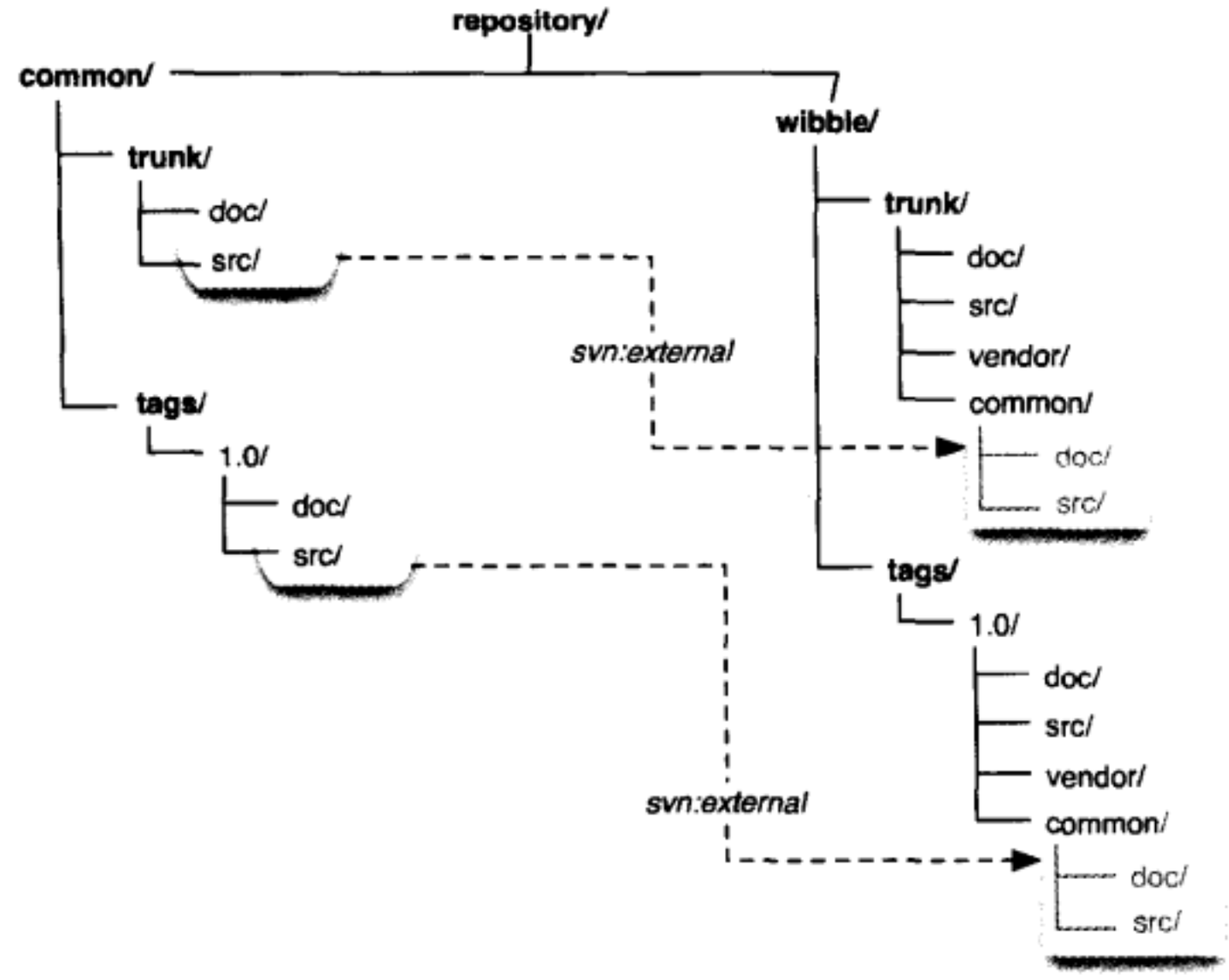


图 10.6 使用外部依赖的项目仓库布局

外部依赖的定义分为两部分：首先在 MaiTai 项目中给 `/dataaccess/trunk` 找个存放的地方。然后提供想要包含的项目仓库 URL。在工作拷贝上执行一次更新，这会导致 Subverison 客户端把数据拉到项目中来：

```
work> cd maitai
maitai> svn update
Fetching external item into 'dataaccess'
A  dataaccess/lib
A  dataaccess/src
Updated external to revision 19.
Updated to revision 19.
```

我们仍然要提交对 `maitai` 目录属性的改动，才能让其他开发者看到新的 external 项。

```
maitai> svn commit -m "Added dataaccess project as an external"
Sending .
Committed revision 20.
```

要做分支的话，外部依赖提供了更多的灵活性。图 10.6 显示 Wibble 项目依赖于 common。而且，Wibble 的 trunk 依赖于 common 的 trunk，但是 1.0 分支的 Wibble 依赖于 1.0 分支的 common。我们可以在 Wibble 项目分支之后通过改变 svn:externals 的定义做到这点。外部依赖还让你可以非常清楚每个项目的依赖有哪些——开发人员不用签出项目仓库中的所有代码就能开始干活。

一个值得注意的地方是当你提交项目时，如果你修改的是签出的外部依赖的代码，Subversion 不会自动提交改动。你须显式地去提交每个外部依赖，通过切换到所在的目录并运行 svn commit，或者在提交的时候把外部依赖目录一个个列出来。

我们推荐在每个项目中把依赖作为“只读”的——如果开发 MaiTai 的开发人员须修正数据访问项目的一个 bug，他应该签出/dataaccess/trunk 来修正这个 bug，再签入，然后在 MaiTai 工作拷贝执行更新以获得修正代码。

第 11 章

第三方代码

Third-Party Code

所有项目都在某种程度上依赖于外部的库：Java 程序使用 `rt.jar`，.NET 程序使用 `mscorlib.dll` 等。这些库应该是你从项目仓库签出的工作拷贝的一部分么？

要回答这个问题，先回答另外一个问题。你需要在将来的任意时间重新构建能工作的程序么？你可以使用这些库将来的版本么？

11.1 二进制库

如果你能肯定在你程序的生命周期内，代码使用的库都能获取到（并且兼容），那么就没有必要对它们进行特殊处理；在你机器上安装使用它们就行了。

在语言提供的标准设施之外，许多项目包含了其他不那么稳定的库。例如，许多.NET 开发者会使用 Nunit 框架[1]来测试他们的代码。比起标准库来，这些框架是相当活跃的（截至至 2006 年 5 月，NUnit 已经升级到了 2.2.8 版了）。虽然版本之间的改动大部分都是兼容的，但是还是可能影响到你的程序。所以我们推荐你把这些库包含到你的项目仓库之中。

[1] http://www.nunit.org/

决定了在工作拷贝和项目仓库之中需要包含第三库之后，你现在需要决定包含什么，以及把它们放哪儿。

第一个决定是包含什么文件。这相对简单。如果你使用的是发布时候原有形式的库，而且你肯定库在程序的生命周期内可以不作修改并正常工作，那么只要存储库的二进制格式就可以了。我们建议把所有的这些库放在你项目的最顶层的 vendor/目录中。

如果库是平台独立的（比如说，Java 的.jar 文件），那么它可以只放在名叫 lib 的子目录中。如果文件是由发布者打包的，在文件名中有版本号，比如 junit-3.8.1.jar，我们建议给它起更通用的名字。在本例中，你把 junit.jar 添加到你的项目仓库之中。这让升级更容易——只要把库的新版本拷贝过来，签入就可以了。你无须更改你的构建脚本或者包含文件。无论是哪种方式，你都应该在提交消息中记录库的版本号，以便将来可以知道你使用的库的版本。

如果你的库是依赖于特定平台的（假定你的程序是可以跨平台的），你就应在 vendor 下给不同的平台和操作系统组建不同的子目录。命名这些子目录的常见方式是使用 arch-os 的格式，其中 arch 指的是目标平台（Intel 的奔腾是 i586，PowerPC 是 ppc，等等）而 *os* 是操作系统（linux，win2k，osx 等等）。

像 C 和 C++这样的语言要求你在使用了特定库的应用程序代码中包含源代码的头文件。这些头文件是库提供的，也应该被存放到项目仓库之中。我们建议你把它们存放在 vendor 下的 include 子目录中。vendor/include 下的目录结构应该以库的头文件能够很自然地被找到为准。举个例子来说，考虑一个叫 *datetime* 的 C 的库的执行日期和时间的计算。它以二进制库包的形式发布，libdatatime.a 和两个头文件，datetime.h 和 extras.h。

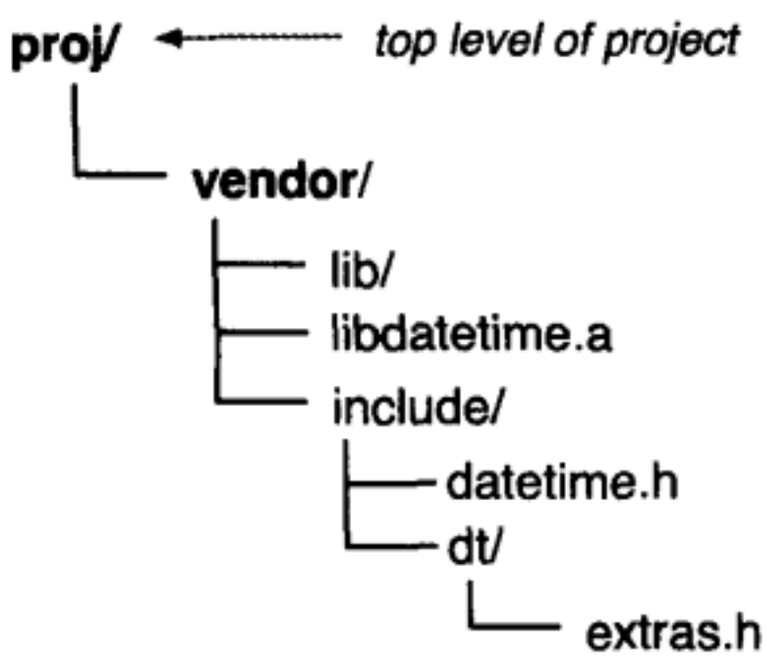


图 11.1 使用第三方库的项目仓库样板

datetime.h 库文件头要放在 include 目录树中的最上层，extras.h 被认为是在 dt 的子目录中。也就是说，使用了这两个程序的开头一般就是这个样子的：

```
#include <datetime>
#include <dt/extras>
// . . .
```

在本例中，我们像图 11.1 那样组织项目仓库（以及工作拷贝）。

■ 与构建环境集成

如果你在项目仓库中包含了第三方的库和头文件，你需要确保你的编译器、链接器和 IDE 能够获取到它们。这有一个小问题：你须确保你没有把任何包含了绝对路径名的东西签入到项目仓库（因为这可能在其他开发者的机器上没法正常工作）。不用绝对路径，你有几个选择：

- 整理你的构建工具，让所有的路径名都是相对于项目目录的根目录。如果你使用的是外部构建工具，像make或者ant，这是可行的，但是需要一些技巧。
- 设置一些外部变量指向项目树的根，然后让构建脚本中所有的引用与变量相联系。这让每个开发者可以设置不同的外部变量，但是共

享同一个构建环境布局。

外部变量不一定要真的是操作系统的环境变量。比如说，Eclipse IDE 让每个用户都可以设置内部的变量，然后有一个公共的构建结构引用这些变量。这意味着所有的开发者可以共享一个Eclipse构建定义，但是每个开发者可以把源代码安装到不同的位置。

我们推荐第二种做法。

11.2 带源代码的库

有时库是带源代码的（或者只发布成源代码的形式）。如果你既有库的二进制版本又有其源代码，你应该在项目仓库中储存哪个呢？你怎么设置你的工作拷贝呢？

答案与风险管理有关。把源代码放在那里意味着你总是可以修正 bug，添加新特性（至少从技术上来讲是如此），有些事情你用二进制库是做不了的。这显然是一件好事情。但是与此同时，把你项目用到的库的源代码包含进去也减慢了构建的速度，并且让项目的结构变得更加复杂，同时也增加了未来的维护负担。如果有一个 bug，他们是不是要考虑改动库的源代码呢，或者把注意力只集中在我们写的代码上？

我们推荐的做法是把第三方的源代码添加到你的项目仓库中，然后特殊对待。要说明如何去做，你需要设身处地去想一下这个问题。

闭上眼睛一分钟，假想你是某个库的作者。你不时会发布和更新代码的版本给你的用户群。作为一个高质量库的作者，你自然会把源代码放在版本控制系统中，并且该做的版本控制的程序都会做。

现在从角色扮演中回来（记住，吸气，呼气，吸气，呼气[2]）。在理想的世界中，我们应该能够直接连接到第三方的项目仓库上，直接从那儿把发布版本拿过来。但是我们不能，所以我们必须自己来做这些工作。无论我们何时从第三方那儿收到代码、bug 修正和新的发布版本，我们必须假设是“我们”产生了这些代码，并且就像它们在它们的版本控制系统所做的那样在我们的版本控制系统中再做一遍。

■ 第一次导入第三方源代码

当我们第一次拿到第三方库的源代码的时候，就须把它们导入到项目仓库中去。第三方的代码被储存在第三方分支中，每次拿到代码，导入其中并称之为 *vendor drop*。我们推荐把第三方的分支和项目自身的代码分离开来。如果你希望一直能从不同的来源导入代码，也许在最顶层目录上建立一个目录更加合适，我们建议称之为/vendorsrc。

vendor branch

vendor drop

每个你想要跟踪的库或者产品在/vendorsrc 下的不同分支中，比如说/vendorsrc/sun/jdbc。在每个第三方分支目录中我们有一个 current 目录存储最新的 vendor drop（有点像常规项目的 trunk 目录）。在同一层，我们有一些标签目录对应于每个 vendor drop。

更具体地说，假设我们决定使用 jMock[3]库的 1.0.0 版本（当然是在检查过了版权协议之后）。

先从 jMock 网站上下载最新的版本。然后把 jmock-1.0.0-src.jar 存储到临时目录，并且使用 WinZip（或者干脆使用原始的 jar 工具）来解开其内容。你应该可以看到一个 jmock-1.0.0 目录包含了所有的 jMock 的源代码，文档和例子。

现在我们可以把这个 drop 导入到项目仓库中。把它储存在/vendorsrc/codehaus/jmock/current。在本例中，厂商是 CodeHaus，“产品”是 jMock。运行 svn import 把 jmock-1.0.0 目录导入：

---

[2] 译注：冷静下来的意思。

[3] http://jmock.codehaus.org/

```
tmp> svn import --no-auto-props -m "Import jMock 1.0.0" \
       jmock-1.0.0                                       \
       svn://olio/vendorsrc/codehaus/jmock/current
Adding       jmock-1.0.0/extensions
Adding       jmock-1.0.0/extensions/cglib
Adding       jmock-1.0.0/extensions/cglib/acceptance-tests
Adding       jmock-1.0.0/extensions/cglib/acceptance-tests/
             atest
Adding       jmock-1.0.0/extensions/cglib/acceptance-tests/
             atest/jmock
         :   :   :   :
Adding (bin) jmock-1.0.0/examples/classes/.../Calculator.class
Adding (bin) jmock-1.0.0/examples/classes/.../ParseException.
             class
Adding (bin) jmock-1.0.0/examples/classes/.../InfixParser.
             class
Committed revision 3.
```

接下来，给 vendor drop 打上标签，标记它为版本 1.0.0。如果 CodeHaus 发布了更新版本的 jMock，你就能很好地跟踪这两个版本了：

```
tmp> svn copy -m "Tag 1.0.0 vendor drop"              \
       svn://olio/vendorsrc/codehaus/jmock/current \
       svn://olio/vendorsrc/codehaus/jmock/1.0.0
Committed revision 4.
```

■ 更新到新的版本

当 jMock 1.0.1 出来了，我们要能够把它加入到项目仓库之中来。在做这件事情之前，让我们变回我们刚才扮演的角色——假设是 CodeHaus，把代码维护在/vendorsrc/codehaus/jmock/current。当发布 1.0.0 时，通过把它拷贝到 jmock/1.0.0 给它打上标签。接着在“主干”上做开发。发布了下一个版本后，打上另外一个标签叫 1.0.1。

实际情况是无法看到对 jMock 代码进行的每个改动的。我们只看到了结果，jmock-1.0.1-src.jar。为了模拟在 jMock 项目仓库所发生的事情，需要更新“我们的”目录/vendorsrc/codehaus/jmock/current 的内容，让它的内容和新的版本一样。更新我们那份 jMock 代码，就好像是我们做了所有的工作让它升级到了 1.0.1 一样。

如何让我们那份代码和新的版本一样呢？因为最新版本的 CodeHaus 可能修改了一些文件，添加了一些文件，甚至把一些文件移动了位置，偶尔也删掉一两个文件。这就须要在 current 中执行所有这些做过的操作。

这样的同步可以手工来做，但是劳动量很大也容易出错。幸运的是，Subversion 有一个实用工具来自动导入新的 vendor drop，帮你完成这些添加和删除操作。这个魔法般的功能是由一个叫 `svn_load_dirs.pl`[4]的 Perl 脚本提供的。

这个脚本要来机器上安装有 Perl，以及一些其它的模块（诸如用来操作 URL 的 URL 模块）。在运行的时候，它需要三个参数：

Base URL

要操作的 Subversion 项目仓库的 base URL。应能够在这个目录下找到某个产品的所有 drop。在我们的例子中，它们是 `svn://olio/vendorsrc/codehaus/jmock`。

“当前”目录

在 base URL 下的这个“当前”目录中可以找到最新的 vendor drop。本例中我们使用 `current`。

要导入的目录

从本机中的这个目录导入新的 vendor drop。

你还可以通过`-t` 选项制定标签名来更新 vendor drop，并自动打好标签。

下载最新版本的 jMock，把它储存在你的临时目录的子目录 `jmock-1.0.1` 中。现在运行 `svn_load_dirs.pl` 载入新的版本并给它打标签：

```
tmp> svn load dirs.pl -t 1.0.1                    \
       svn://olio/vendorsrc/codehaus/jmock current jmock-1.0.1
Directory jmock-1.0.1 will be tagged as 1.0.1
Please examine identified tags. Are they acceptable? (Y/n) y
```

我们被询问说是否把 `jmock-1.0.1` 的新源代码打标签为“1.0.1”。那也是我们想做的，所以输入 y 并回车。之后的过程都是自动的：

```
Checking that the base URL is a Subversion repository.
Running /usr/local/bin/svn log -r HEAD svn://olio/vendorsrc/
codehaus/jmock
Finding the root URL of the Subversion repository.
Running /usr/local/bin/svn log -r HEAD svn://olio
Determined that the svn root URL is svn://olio.
```

---

[4] http://svn.collab.net/repos/svn/trunk/contrib/client-side

```
Native EOL on this system is \012.
Finding if any directories need to be created in repository.
Running /usr/local/bin/svn log -r HEAD svn://olio/.../jmock/current
No directories need to be created to prepare repository.
Checking out svn://olio/.../jmock/current into /tmp/...
Running /usr/local/bin/svn checkout svn://olio/.../jmock/current my
import wc
Loading jmock-1.0.1 and will save in tag 1.0.1.
U  build.properties
U  VERSION
U  CHANGELOG
U  core/src/test/jmock/core/InvocationTest.java
U  core/src/test/jmock/core/testsupport/MockInvocationMatcher.java
U  core/src/test/jmock/core/matcher/InvokedRecorderTest.java
        :                 :                    :
Running /usr/local/bin/svn propget svn:eol-style VERSION
Running /usr/local/bin/svn propget svn:eol-style CHANGELOG
        :                 :                    :
Running /usr/local/bin/svn commit --file /tmp/svn load ...
Running /usr/local/bin/svn update
        :                 :                    :
Cleaning up /tmp/svn load dirs ZH6k9TLxFM
```

检查 Subversion 的日志，可以看到 jMock 1.0.0 和 1.0.1 之间的改动已经反映到本地的代码中了：

```
tmp> svn log -v svn://olio/vendorsrc/codehaus/jmock/current
------------------------------------------------------------
r5 | mike | 2004-11-18 17:03:06 -0700 (Thu, 18 Nov 2004)
Changed paths:
     M /vendorsrc/codehaus/jmock/current/CHANGELOG
     M /vendorsrc/codehaus/jmock/current/VERSION
     M /vendorsrc/codehaus/jmock/current/build.properties
          :                 :                    :
Load jmock-1.0.1 into vendorsrc/codehaus/jmock/current.
------------------------------------------------------------
```

当有新版本的 jMock 出来的时候，可以执行同样的过程。

■ 在项目中使用第三方代码

现在，导入已经被很漂亮地完成了——你已经有了自己的一份 jMock 源代码并且为了日后的使用，给它们打了标签。现在要在项目中实际使用这些源代码。为了使用它们，可把第三方分支拷贝到你的项目中，储存在 vendor/jmock 下：

```
work> svn mkdir -m "" svn://olio/maitai/trunk/vendor
work> svn copy -m "MaiTai needs jMock"                          \
                  svn://olio/vendorsrc/codehaus/jmock/1.0.0 \
                  svn://olio/maitai/trunk/vendor/jmock
Committed revision 12.
```

当我们签出 MaiTai 时，同时获得了一份 jMock 代码：

```
work> svn checkout svn://olio/maitai/trunk maitai
A  maitai/doc
A  maitai/src
A  maitai/vendor
A  maitai/vendor/jmock
A  maitai/vendor/jmock/extensions
      :    :    :    :
A  maitai/vendor/jmock/build.xml
Checked out revision 12.
```

■ 修改第三方代码

现在你有了第三方的源代码，可以自由对其进行修改，你的改动可以很容易地合并到新的版本中去。

比如说我们想要对 jMock 的异常处理和设置值的预期框架做一些处理。在 MaiTai 工作拷贝中做你的修改，然后如往常一样提交：

```
maitai> svn status
M       vendor/jmock/core/src/org/jmock/expectation/
        ExpectationList.java
M       vendor/jmock/core/src/org/jmock/util/NotImplemented
        Exception.java
maitai> svn commit -m "Made some custom changes to jMock"
Sending  vendor/jmock/core/src/org/jmock/expectation/
         ExpectationList.java
Sending  vendor/jmock/core/src/org/jmock/util/NotImplemented
         Exception.java
Transmitting file data ..
Committed revision 13.
```

Subversion 对待它们就像对待你平时所写的代码一样，跟踪了所有做过的改动。

■ 更新修改过的代码

生活是美好的。MaiTai 项目进展得很顺利并且对于你的公司来说是成功的。CodeHaus 的人们发布了新版本的 jMock，并且你想要把它集成进 MaiTai 项目中来。在载入了新的 vendor drop 并打好了标签之后，你就可以开始升级 MaiTai 了。

我们要合并 jMock 在 1.0.0 和 1.0.1 之间的改动。可以使用 `svn merge` 命令来做这件事情：

```
maitai> svn merge svn://olio/vendorsrc/codehaus/jmock/1.0.0 \
         svn://olio/vendorsrc/codehaus/jmock/1.0.1           \
         vendor/jmock
U  vendor/jmock/VERSION
U  vendor/jmock/CHANGELOG
      :       :       :
U vendor/jmock/build.properties
```

Subversion 把改动应用到你的工作拷贝。如果在你的改动和 1.0.1 的改动之间有任何冲突的话，你要像平时两个开发人员一起开发那样去修正冲突。一旦所有的冲突都解决好了，在运行了测试并且确信所有东西都一切正常，就可以把改动提交给项目仓库了：

```
maitai> svn commit -m "Updated MaiTai with jMock 1.0.1"
Sending        vendor/jmock/CHANGELOG
Sending        vendor/jmock/VERSION
Sending        vendor/jmock/build.properties
                    :       :       :
Transmitting file data .....................
Committed revision 14.
```

11.3 在导入过程中的关键字展开

在这些例子中，我们把第三方代码（可能不是从 Subversion 的版本控制系统那里导入的）导入到了项目仓库中。比如说，如果我们是从 CVS 导入的代码，那么作者可能包括了`$Author$`或者`$Id$`这样的关键字。我们在第 71 页的 6.4 节“关键字展开”中对关键字有完整的讨论。

关键字的问题是每次签出的时候都会被展开。如果第三方使用了这些标签，那么你拿到的源代码可能在这些域中有第三方厂商的信息。然而，如果你只是原样导入这些文件然后再签出，Subversion 会更新标签，突然你的名字就出现在作者那一栏中了。也许这不会是啥问题，但是当你在下一个版本发布后去合并改动的时候就会遇到问题。Subversion 会注意到这些标签所在的行已经被改动了，你会在合并第三方代码时发生冲突。

幸运的是，默认情况下关键字是不会应用于新文件的。然而，如果你像第 77 页的 6.4 节“自动属性设置”中所描述的那样设置了自动属性，关键字展开可能就会发生。在导入的时候可以使用`--no-auto-props`开关来禁用可能的关键字展开。

使用分支来处理第三方代码的 SCM 模式叫做“third party codeline”。

附录 A

Subversion 的安装，联网，安全和管理

Install, Network, Secure, and Administer Subversion

Subversion 客户端的安装是相当简单的，一般只要找到适合你操作系统的版本，下载下来就可以了。运行一个服务器要更复杂一点，而且许多人，特别是那些从 CVS 迁移过来的人，想要把 Subversion 服务器运行在 Unix 平台上。Subversion 的数据库后台还需要一个不同的备份策略而不只是普通的基于文件的版本控制系统。本附录讲述了如何在 Windows 和 Linux 下安装 Subversion，让你的项目仓库联网，以及如何去做备份以防万一。除此之外还讨论了如何使得你的项目安全，让那些窥探的人什么东西也看不到。

A.1 安装 Subversion

Subversion 给不同的操作系统打包有不同的版本。[1] 如果你使用的是基于 Unix 的系统，Subversion 可能有官方的打包版本，所以先使用你的包管理器看看是不是有官方版本。

■ Windows 安装

友好的 Windows 安装程序让 Subversion 安装起来很便捷，甚至把二进制

[1] 去 http://subversion.tigris.org/project_packages.html 查看各种打包版本的完整列表。

数据都放到你的 PATH 环境变量中了。如果你还打算安装 Apache，请在安装 Subversion “之前”安装它。这样的话，Subversion 安装工具会自动拷贝 Subversion 的 Apache 模块到正确的位置。

■ Linux 安装

这儿我们来讲讲如何在 Fedora Core 5 下安装，它有一个标准打包版的 Subversion 1.3。你既可以使用 Fedora 的包管理器安装，也可以手工地去下载。

要使用 GUI 的包管理器，选择 *Applications > Add/Remove Software*。在 “Servers”分类中，确保“Web Server”被选中。在“Development”下，选择“Development Tools”然后选中可选包按钮把 Subversion 选中。点击 Update 按钮让改动生效。

要从命令行安装，使用 yum 包管理器：

```
root> yum install httpd subversion mod dav svn
Dependencies Resolved
================================================================
Package              Arch  Version        Repository   Size
================================================================
Installing:
 httpd               i386  2.2.0-5.1.2    core         1.1 M
 mod_dav_svn         i386  1.3.0-4.2      core         65 k
 subversion          i386  1.3.0-4.2      core         2.1 M
Transaction Summary
================================================================
Install              3 Package(s)
Update               0 Package(s)
Remove               0 Package(s)
Total download size: 3.3 M
Is this ok [y/N]:
```

yum 告诉你一声他要去安装你所要它安装的包了，除此之外，还有两个 Subversion 所需要的包 apr 和 apr-util。在下载完成之后，你可以看到这样的屏幕：

```
Running Transaction
  Installing: httpd          ########################### [1/3]
  Installing: subversion     ########################### [2/3]
  Installing: mod_dav_svn    ########################### [3/3]
Installed: httpd.i386 0:2.2.0-5.1.2 mod_dav_svn.i386
           0:1.3.0-4.2 subversion.i386 0:1.3.0-4.2
Complete!
```

说明 Subversion 已安装好，可以开始用了。

A.2 使用 svnserve 联网

svnserve 是 Subversion 的简易版本的联网服务器。它速度快使用简单，适合在流量是保密的无窃听可能的公司局域网内使用。

■ 在 Windows 上运行 svnserve

要在 Windows 上启动 svnserve，需要在你的命令行提示符下输入：

```
C:\> start svnserve --daemon --root c:\svn-repos
```

一个新的窗口会打开，标题是 svnserve.exe。如果你使用的是 Windows XP 或者安装了其它的防火墙软件，你可能会被询问是否允许这个服务器建立网络连接，此时选择 unblock svnserve。我们用 --daemon 选项让 svnserve 以后台进程模式启动（Windows 实际上并没有以后台进程形式运行；这个选项不过是让你的 svnserve 启动时必须要指定的一个古怪选项），我们用 --root 参数选择了允许访问的项目仓库名。

弹出一个新窗口并不好，因为你可能偶然地把它关了。如果你想要 svnserve 运行时没有自己的窗口，可以在 start 命令之后添加 /B。但是这样的话，你需要使用任务管理器才能在它完成任务时关掉它。

如果你想要 svnserve 在你 Windows 服务器启动的时候自动运行，就要把它安装成服务器。Magnus Norddahl 维护了一个简单的 service wrapper 称作 svnservice，可以从 http://dark.clansoft.dk/~mbn/svnservice/ 下载到。

■ 在 Unix 上运行 svnserve

在 Unix 上启动 svnserve 与上述非常类似：

```
home> svnserve --daemon --root /home/mike/svn-repos
```

你的命令行直接返回了，svnserve 作为后台进程运行着。运行 ps 应该可以看到仍旧在运行的进程。

尝试着从你网络上的不同的机器访问项目仓库。本书中的例子服务器称作 olio，所以你可以运行：

```
work> svn co svn://olio/sesame/trunk vizier
A  vizier/Number.txt
A  vizier/Day.txt
Checked out revision 7.
```

如果这样用不行，你可能要检查一下在两台机器之间是否有防火墙。如果有（比如说，ZoneAlarm，Windows XP 内建的防火墙，或者 Unix 防火墙），就须确保运行 svnserve 的机器可以接收来自 TCP 端口 3690 的连接。

设置好这些之后，你就应该考虑如何保证项目仓库的安全了。因为默认的 svnserve 允许匿名用户只读访问一切内容。参考第 169 页的 A.5 节“保证 Subversion 的安全”，那里有详细的指导。

A.3 使用 svn+ssh 联网

Windows 一般不支持进入的 SSH 连接，所以这里只讲 Unix 的配置。你可能也可以在 Windows 上把 Putty 当 SSH 服务器使，但是这绝对不适合心脏病患者！

当用户指定 svn+ssh 方式来访问项目仓库的时候，Subversion 客户端运行 SSH 来连接到服务器。这意味着每个用户要在服务器上有一个账号，并且输入的密码是 Unix 账号的密码。如果你的用户有一个公匙密匙对或者运行着 SSH 代理，Subversion 会自动利用这些特性。

Subversion 尝试着在服务器运行 svnserve -t 以访问项目仓库。如果 Subversion 报错说它找不到 svnserve，确保服务器上默认的 PATH 中包含了 svnserve 可执行文件。因为 Subversion 使用 -t（tunnel）选项启动 svnserve，你没有必要像使用普通的 svn 连接时那样把它运行成后台进程。

一旦 SSH 连接建立好了，svnserve 运行在 tunnel 模式，Subversion 会尝试着访问项目仓库的文件。它做这件事时是以通过 SSH 验证的用户的身份去做的，这意味着你的项目仓库的所有用户须有对项目仓库中文件的读写

权限。除此之外，任何创建出来的“新”文件对其它用户来说应是可读可写的。[2]

为了让多个 Unix 用户访问项目仓库，他们应该在同一个 Unix 组之中，并且使用 SSH 运行 `svnserve` 的时候设置 umask 为 002。你还需要在项目仓库目录设置组的“sticky bit”。这儿有一个一步步地指导告诉你如何设置好这些。

首先给所有使用 Subversion 的人创建一个 Unix 组，并且把每个用户都添加到这个组之中。这些命令是针对 Linux 的，对于你使用的 Unix 可能需要稍微调整一下才能用：

```
root> /usr/sbin/groupadd subversion
root> /usr/sbin/usermod -G subversion mike
root> /usr/sbin/usermod -G subversion ian
```

接下来，改变你的项目仓库目录以及文件的所有者为刚才创建的组，并且为项目仓库的 db 目录设置组的 sticky bit：

```
root> chgrp -R subversion /home/svn-repos
root> chmod -R 770 /home/svn-repos
root> chmod g+S /home/svn-repos/db      # g+t on BSD systems
```

现在尝试着从项目仓库签出文件。这儿我们要额外地输入一个远程机器上的用户名以及对应的 Unix 密码：

```
work> svn checkout                                                \
          svn+ssh://mike@olio/home/svn-repos/sesame/trunk \
          sesame
mike@olio's password:
A  sesame/Number.txt
A  sesame/Day.txt
Checked out revision 7.
```

这有一点复杂，但是如果你想要利用 SSH 以确保连接安全的话还是值得的。在网络上还有更多信息可供查阅。[3]

---

[2] 这事没做对是导致项目难以琢磨地死掉的常见原因。在提交过程中，BDB 会创建新文件，这些文件也是项目仓库的一部分。如果它们对于其它用户是不可写的，他们的 Subversion 客户端在尝试着访问项目仓库的时候会死掉。

[3] http://svnbook.red-bean.com/en/1.1/ch06s03.html#svn-ch-6-sect-3.4

■ SSH 常见问题

使用 svn+ssh 连接到项目仓库时，须配置不少程序。不幸的是，Subversion 的错误信息有时不是那么有用。这有一个粗略的指南，针对可能出错的东西，告诉你如何去修正它们。

■ svn:系统不能找到指定的文件（Windows）

Subversion 报错说它找不到“指定的文件”。在这种情况下，它是在寻找 ssh 以创建安全连接（svn 命令已经被找到了）。通常的修正办法是编辑你的 Subversion 配置如前面第 59 页的 5.1 节“*svn+ssh*”所述，把 plink.exe 放到你的 PATH 之中。

■ svn:没有这样的文件或者目录（Unix）

类似于 Windows 的“不能找到指定的文件”问题，Subversion 不能在你的系统中找到 ssh 命令。这可能意味着 ssh 在你的机器上没有安装。

■ Subversion 看起来就像死了一样（Windows）

打开任务管理器，看看 plink.exe 是不是在运行。如果它在运行，但是 Subversion 没有显示任何输出，这可能是因为 plink 在等待用户输入。这在你第一次连接到 SSH 服务器的时候可能发生，此时需要接收一个服务器 key。尝试着独立运行 plink，在询问的时候告诉它“yes”以把 key 存储在 Putty 的缓存之中：

```
work> plink mike@olio.mynetwork.net echo hello
The server's host key is not cached in the registry. You
have no guarantee that the server is the computer you
think it is.
The server's rsa2 key fingerprint is:
ssh-rsa 1024 c3:82:fd:a6:b4:5d:23:f2:1a:f8:8b:04:be:c3
If you trust this host, enter "y" to add the key to
PuTTY's cache and carry on connecting.
:         :         :
Store key in cache? (y/n) y
mike@olio.mynetwork.net's password:
hello
```

■ svnserve：命令没有找到

■ svn：连接异常关闭

Subversion 打印了两条出错信息中的一行或者两条时，表明它在服务器上找不到 svnserve。SSH 连接已经被建立并且你已经被验证为一个 Unix 用户了，但是 svnserve 不在用户的 path 之中。

不幸的是，Subversion 总是尝试着用 svnserve -t 在远程服务器上运行，所以你不能通过告诉客户端 Subversion 安装在哪儿解决这个问题。你必须改变服务器的默认 path，可能是编辑/ect/profile。一旦你把 svnserve 放在 path 之中了，就能够从客户端进行这样的测试来验证是否解决问题了：

```
work> ssh mike@olio.mynetwork.net svnserve -t
( success ( 1 2 ( ANONYMOUS EXTERNAL ) ( edit-pipeline ) ) )
```

括号中的成功消息是在 Subversion 客户端和服务器之间建立的 svn 协议的协议头，这意味着 svnserve 已经被正确地找到了。

■ svn：在‘svn+ssh:// myserver/home/ svn-repos’没有找到项目仓库

Subversion 已经成功地使用 SSH 连接并且启动了 svnserve。然而，svnserve 不能找到项目仓库。检查你使用了正确的项目仓库路径并且你在项目仓库目录中有足够的权限读取并创建文件。

A.4 使用 Apache 联网

在本节中我们将展示如何安装 Apache 并且把它配置成为 Subversion 项目仓库服务的模式。Unix 安装的方法对于不同的 Unix 版本稍微有些不同，但是在网络上有教你怎么做的地方。我们使用 Fedora Core5 作为 Unix 平台的例子。

Subversion 开发者自己写的 Subversion 书可能是最好的完整版参考书，可以在 http://svnbook.red-bean.com/[CSFP]中找到。

Subversionary 网站[4]的“How-To”包含了数个操作系统的联网指南，包括 RedHat 和 Windows。

■ 在 Windows 上使用 Apache

■ 下载并安装 Apache

Apache是开源软件，你可以免费从`http://httpd.apache.org/download.cgi`下载到。

对于 Windows 下安装，你可以下载`.exe`或者`.msi`文件。MSI 是 Windows 安装程序包并且尺寸更小，所以最好选择这个。Subversion 至少需要 Apache 版本 2.0.48——在本例中，我们使用的版本是 2.0.50。

运行安装程序，阅读版权协议和安装说明，然后你可以看到如图 A.1 所示的屏幕。在屏中填入正确的信息，要不然用户在连接 Apache 的时候就会遇到麻烦。如果你不确定使用什么样的设置，可询问你的网络管理员。

我们只给当前用户安装 Apache，安装到的端口是 8080。Windows 机器经常已经有一个运行着的服务器在普通的 HTTP 端口 80 上了，我们不想让 Apache 服务器与之冲突。如果你想把 Subversion 项目仓库设置为使用 80 端口，确信 Internet Information Services（IIS）被关闭了。

下一步选择典型安装，并且使用 Apache 安装的默认目录。你可以看到在 Apache 安装同时一些命令窗口弹出来了，跟着有一条消息告诉你安装成功了。

此时，Aapche 没有运行，因为我们在安装的时候选择了“just for the current user”选项。要启动 Apache，可选择 *Start > All Programs > Apache HTTP Server 2.0.50 > Control Apache Server > Start Apache in Console*。一个命令行

---

[4] `http://www.subversionary.org/`

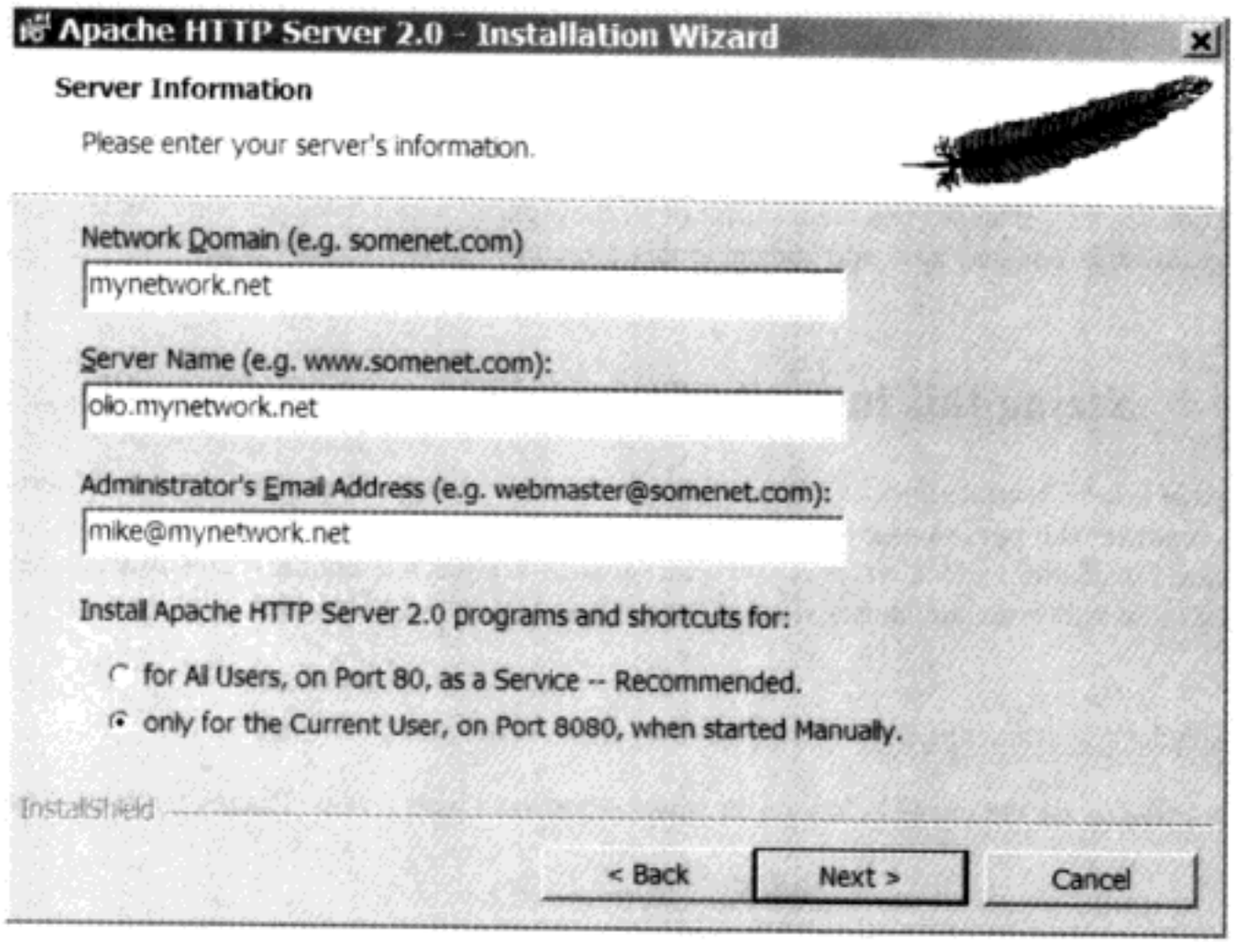


图 A.1 Apache 服务器名配置

窗口会出现，这意味着 Apache 已经在运行了。[5]打开 web 浏览器指向 `http://localhost:8080` 你可以看到如图 A.2 所示的测试页。

■ 安装 Subversion 的 Apache 模块

Subversion 使用数个二进制模块与 Apache 集成，它们需要安装在正确的位置才能发挥作用。

如果你使用的是 Windows，打开 `C:\Program Files\Subversion\httpd` 并且拷贝 `mod_authz_svn.so` 和 `mod_dav_svn.so` 到 `C:\Program Files\Apache Group\Apache2\modules` 目录。然后切换到 `C:\Program Files\Subversion\bin`，拷贝文件 `libdb42.dll` 到 `C:\Program Files\Apache Group\Apache2\bin`。如果你在安装 Subversion 之前已经安

[5] 我们本来打算在书中包含了一个 Apache 控制台窗口的截图的，但是 Dave 认为这看起来有点像 “Bournemouth by Night” 的笑话贺卡，所以就没放进来。不要担心当 Apache 窗口打开的时候里面啥也没有——它们应该就是那样子的。

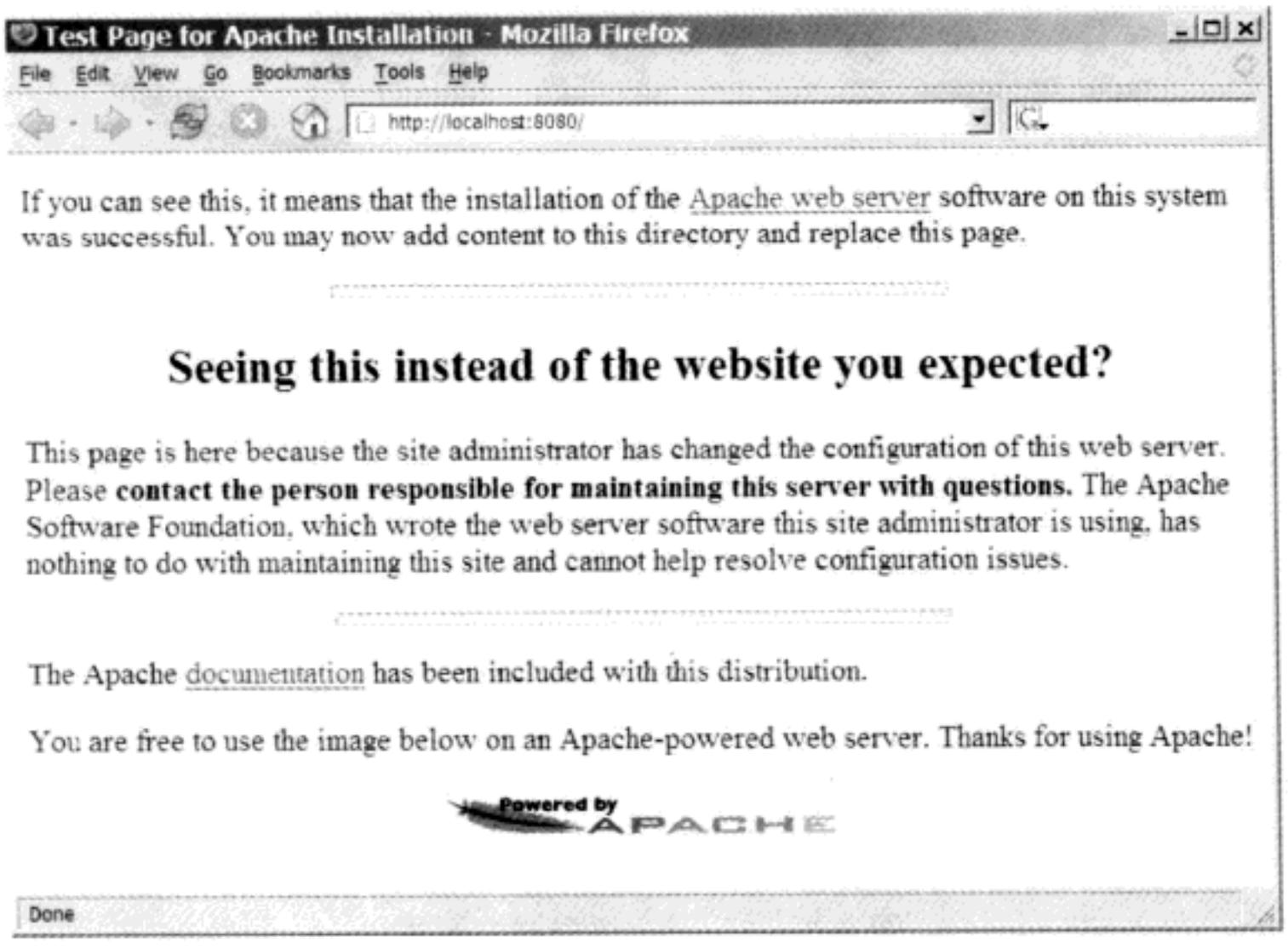


图 A.2 Apache 测试页

装好了 Apache，Subversion 的安装程序会在安装 Subversion 的时候为你做好这些拷贝工作。

■ 配置 Apache

配置 Apache 须编辑你的 Apache 安装位置中的`.conf`文件。要编辑的文件在不同的系统上稍微有一些不同——Windows 使用一个`httpd.conf`，而且在不同的 Unix 版本也会有所不同，Red Hat Linux 使用`conf.d`目录中的数个小文件来代替单一的大文件。

选择 *Start > All Programs > Apache HTTP Server 2.0.50 > Configure Apache Server > Edit the Apache httpd.conf Configuration file*。这样会打开记事本来编辑 Apache 配置文件。

往下滚到“Dynamic Shared Object (DSO) Support”节。你可以看到大量的 LoadModule 命令，每个都激活了一个 Apache 的扩展功能。在列表的最底下，添加两个新行：

// Joe 问……
DAV，WebDAV，DeltaV

作为与 Apache 集成的一部分，Subversion 使用 WebDAV 作为客户端和服务器之间的协议。WebDAV 指的是“Web-based Distributed Authoring and Versioning”（基于 web 的分布式授权和版本记录）而且它是 HTTP 协议的一个扩展。Subversion 的开发者决定利用 WebDAV 而不是从头开发自己的网络协议。

重用带来了大量的好处，既有开发速度方面的，也有与其他客户端兼容方面的。例如，Windows 和 Mac OS X 都能与 WebDAV 服务器连接，把它作为一个网络驱动器。

要获得关于 WebDAV 的更多信息，包括客户端配置方面的，可参见 http://www.webdav.org/。

```
LoadModule dav_svn module modules/mod_dav_svn.so
LoadModule authz_svn_module modules/mod_authz_svn.so
```

接下来，往上滚动一点，取消掉 dav_module 那行的注释：

```
LoadModule dav_module modules/mod_dav.so
```

最后，滚动到文件的底部，添加下面这个节：

```
<Location /svn-repos>
    DAV svn
    SVNPath c:\svn-repos
</Location>
```

这告诉 Apache 由 /svn-repos 开头的 URL 应该使用 Subversion DAV 模块并且这个项目仓库位于 c:\svn-repos。

如果 Apache 仍然在运行，则关闭命令窗口，把服务停掉。然后在控制台菜单项重启 Apache，并浏览 http://localhost:8080/svn-repos/。如果 Subversion 和 Apache 被正确配置了，你就可以看到你的项目仓库和 Sesame 项目，如图 A.3 所示。

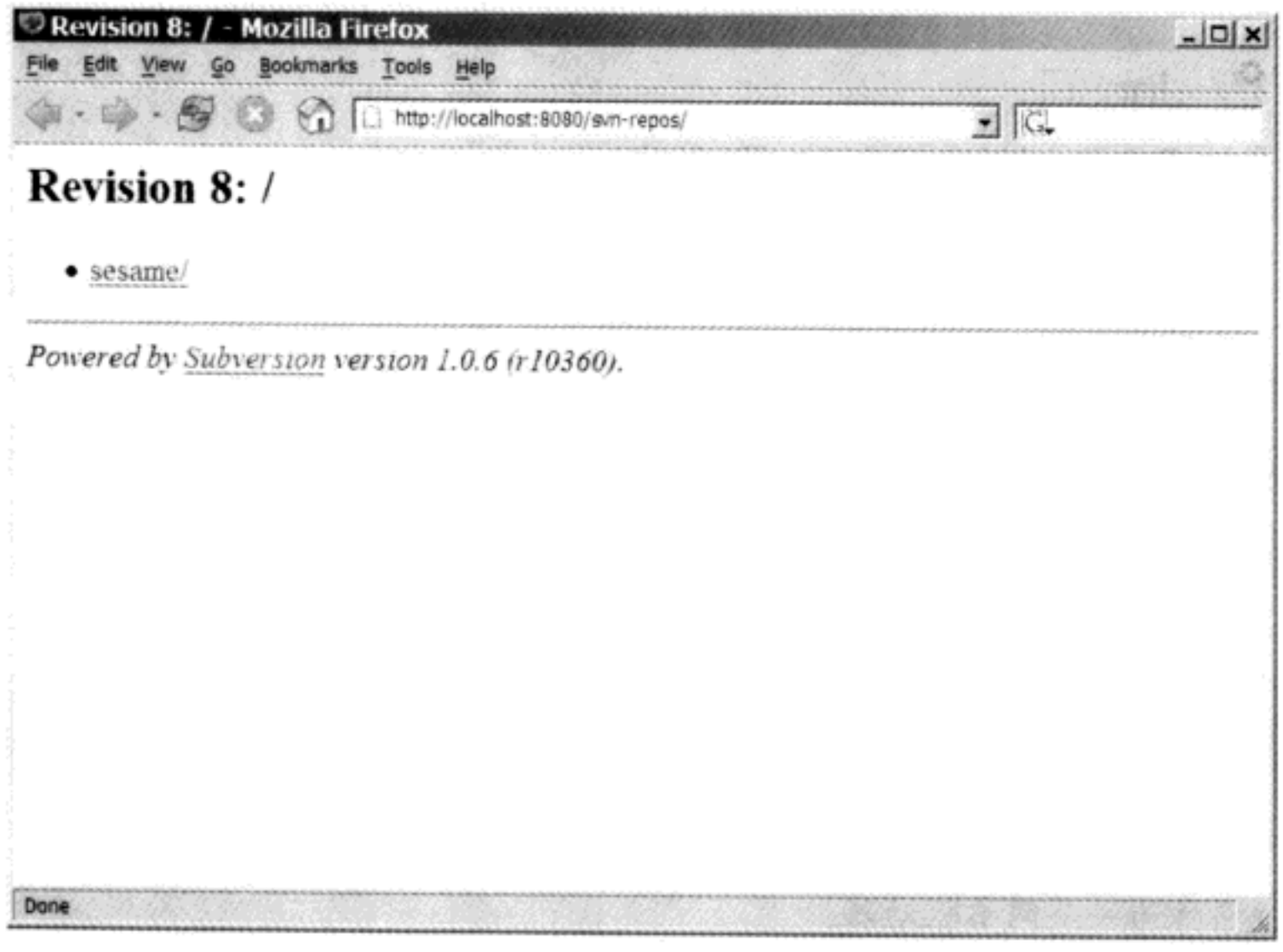


图 A.3 浏览 Subversion 项目仓库

试着在项目仓库中转转——点击文件会显示最后签入的内容，点击目录会让你进入那个目录。

■ 在 Red Hat Linux 上使用 Apache

Fedora Core 5 通常安装的时候就自带了 Apache。如果你跟着本章中的安装指导去做了，可能已经安装好了 Apache，Subversion，以及 Apache 的集成模块 `mod_dav_svn`。如果没有，打开你的包管理器把它们装上。

■ 配置 Apache

在 Red Hat Linux 上，Apache 使用了数个`/etc/httpd/conf.d` 中的 conf 文件。安装 `mod_dav_svn` 添加了一个新的 `subversion.conf` 文件到这个目录中，你须编辑它以让它指向你的项目仓库，其他版本的 Unix 还是只有一个 `httpd.conf` 文件放在`/etc/httpd` 之中。

subversion.conf 的内容暗示其安全设置要到第 171 页的 A.5 节，“Apache 安全性”中才能讲到，但是对于现在只要把这些行取消掉注释，让其指向你的项目仓库就够了：

```
<Location /svn-repos>
    DAV svn
    SVNPath /home/svn-repos
</Location>
```

确信 svn-repos 目录被 Apache 用户所拥有，并重启 Apache web 服务器：

```
root> chown -R apache /home/svn-repos
root> service httpd restart
Stopping httpd:                            [ OK ]
Starting httpd:                            [ OK ]
```

此时你的项目仓库是不安全的，对于匿名用户可以读写。不要让它一直是这个样子！第 171 页的 A.5 节“Apache 安全性”，对于如何保证你的 Apache 伺服的项目仓库安全有着详细的介绍。

A.5 保证 Subversion 的安全

当要访问 Subversion 的项目仓库的时候，安全性主要反映在两个方面：用户身份验证和基于路径的授权。用户身份验证是确保连接到项目仓库的人已经被授权；基本上来说就是用密码来保护你的数据。任何人提供了一个有效的用户名和密码都被授权可以访问项目仓库。基于路径的安全则要更进一步，在不同的用户之间，允许或者拒绝对项目仓库中某些目录的访问。

■ svnserve

默认设置下，svnserve 是只读的项目仓库。要获得读和写的访问权，必须编辑 svnserve.conf，位于 svn-repos/conf 目录。当你创建项目仓库的时候，svnserve.conf 看起来就如图 A.4 一样。

如果你熟悉.conf 文件，可以看到整个文件都被注释掉了（以#开头的行是注释）。Subversion 想要给用户提供一些帮助，给了我们一些关于如何配

置的提示，但实际上大部分人都会被这善意的帮助搞糊涂。

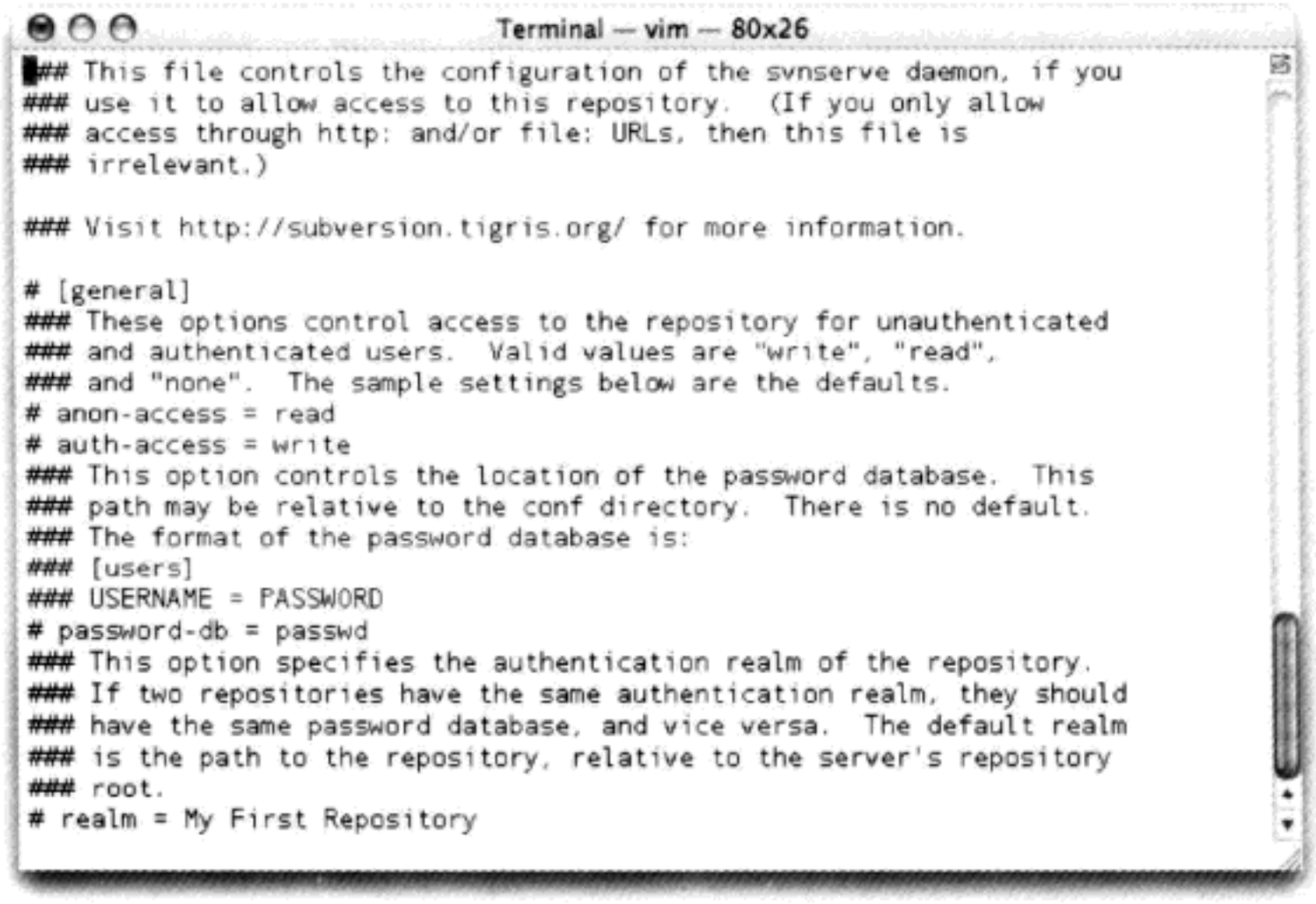


图 A.4 默认的 svnserve.conf

让我们忽略掉默认的配置，从头创建一个简单的 `svnserve.conf`：

svnrepos/conf/svnserve.conf

```
[general]
anon-access = read
auth-access = write
password-db = passwd
```

这样就告诉 `svnserve` 了允许匿名用户只读访问项目仓库，以及对验证了身份的用户可以读/写访问。我们还告诉 `svnserve` 在 `passwd` 文件中寻找用户名和密码。在同一个 `conf` 目录下，创建像这样的密码文件：

svnrepos/conf/passwd

```
[users]
mike=secret
dave=n1nja123
ian=b4n4n4
```

我们已经定义了三个用户，每个都有自己的密码。要提交一个改动到项目仓库，客户端必须提供一个有效的用户名和密码。

如果你使用的是 Subversion 1.3，`svnserve` 可以提供基于路径的安全性，使用和 `mod_authz_svn` 同样的安全配置文件。如果你使用的是 Subversion

更早一些的版本，或者没有配置基于路径的安全性，用户可以读写访问整个项目仓库。

要启用基于路径的安全性，把以下代码添加到你的 svnserve.conf 文件中去：

```
authz-db = authz
```

接下来，创建授权数据库 authz，如第 173 页的 A.5 节“基于路径的安全性”所述。

■ svn+ssh

连接到使用 svn+ssh 的项目仓库，使用 Unix 的安全性决定一个用户是否有权访问一个项目仓库。如果他们能够访问项目仓库的数据库文件，他们就有了对项目仓库的读写权限。

如 svnserve 一样，使用 svn+ssh 时也可以使用钩子脚本达到基于路径的安全性。还可以伺服多于一个的项目仓库在同一台 Unix 服务器上，使用不同组的用户授权给不同的项目仓库，同样使用的是标准的 Unix 权限。

■ Apache 安全性

如果你一直跟着本章的内容做，你已经有了一个使用基本配置的 Apache 的在线项目仓库。当你使用这样的设置时，项目仓库给予每个人读和写的权限，包括匿名用户在内。

我们最好赶紧修复这个问题。

Apache 给用户提供了大量的身份验证方式。这儿我们只设置基本的密码验证，但是你可以做更复杂的配置，比如用 Windows 域来验证身份。基本的验证需要密码文件，其中包含你的用户名和密码，而且你要用 Apache 中的实用工具 htpasswd 来创建它。

如果你在 Windows 上，打开命令行，把当前目录切换到 C:\Program Files\Apache Group\Apache2\bin，运行：

```
bin> htpasswd -c -m c:\svn-repos\conf\htpasswd mike
New password: ******
Re-type new password: ******
Adding password for user mike
```

在 Unix 上，htpasswd 应该已经在你的 path 中了，所以可直接运行：

```
home> htpasswd -c -m /home/svn-repos/conf/htpasswd mike
New password: ******
Re-type new password: ******
Adding password for user mike
```

一旦文件创建好了，你可以通过-c 选项给它添加新的用户：

```
bin> htpasswd -m c:\svn-repos\conf\htpasswd dave
New password: ********
Re-type new password: ********
Adding password for user dave
```

接下来我们要让 Apache 在用户访问项目仓库之前验证其身份。我们可以对所有的操作都要求对方是有效用户，或者只对那些实际修改项目仓库的人才这么要求（也就是说匿名者可以访问）。为了完全地把东西锁上，应把你的 Apache 路径配置修改成这样：

```
<Location /svn-repos>
DAV svn
SVNPath c:\svn-repos
AuthType Basic
AuthName "Subversion Repository"
AuthUserFile c:\svn-repos\conf\htpasswd
Require valid-user
</Location>
```

要让匿名可以只读访问，可把 Apache 配置成这样：

```
<Location /svn-repos>
DAV svn
SVNPath c:¥svn-repos

AuthType Basic
AuthName "Subversion Repository"
AuthUserFile c:\svn-repos\conf\htpasswd

<LimitExcept GET PROPFIND OPTIONS REPORT>
    Require valid-user
</LimitExcept>

</Location>
```

重启 Apache 之后你对配置的改动就会生效了。

如果你的 Apache 服务器有 SSL 证书，你可以要求访问项目仓库的时候需要有一个安全连接。这样就会加密在 Subversion 客户端和项目仓库之间的流量，包括密码和文件内容，而且如果你想要让你的项目仓库可以在互联网

上访问的话，这是一个不错的选择。再次编辑你的 Apache 配置，添加

```
SSLRequireSSL:

        <Location /svn-repos>
        DAV svn
        SVNPath c:\svn-repos

        AuthType Basic
        AuthName "Subversion Repository"
        AuthUserFile c:\svn-repos\conf\htpasswd
        Require valid-user
        SSLRequireSSL
        </Location>
```

■ 基于路径的安全性

svnserve 和 Apache 都可以配置为使用基于路径的安全性，使用它可以把访问限制在项目仓库中的某些目录。这个可以通过设置 mod_authz_svn（对于 Apache 而言）或者 authz_db（对于 svnserve 而言）的配置文件来完成。两者使用相同的文件格式来定义授权数据库。

要在 Apache 中启用授权数据库，编辑你的 subversion.conf 文件，并添加以下小节到你的项目仓库定义中：

```
AuthzSVNAccessFile c:\svn-repos\conf\authz
```

授权数据库文件包含了组定义和路径安全性定义。[groups]配置小节命名了组以及定义了组内有哪些用户。基于路径的安全性定义把访问项目仓库的路径的访问权限关联到了用户或者用户的组。授予的权限可以是只读的，可读可写的，或者不可访问，分别对应于“r”，“rw”或者“ ”。

举例来说，假设我们有两个开发者 Fred 和 Wilma，他们都能够读写 Sesame 项目树，其余人只能读取这个树。authz 文件应该像这样：

```
[groups]
developers = fred, wilma

[/projects/sesame]
@developers = rw
* = r
```

新建立的，顶级机密的“Project Blue”只能被我们最聪明最忠诚的 Barney 访问。我们需要以下的安全性定义：

svnrepos/conf/authz

```
[/projects/blue]
barney = rw
* =
```

安全性权限是从父目录继承到子目录的。你通过包含一个更具体的安全性定义的办法来加强（或者放松）权限。为了给测试组提供对 Sesame 的 testing 目录的写权限，授权文件应该这么写：

```
[/projects/sesame]
@developers = rw
* = r
[/projects/sesame/testing]
@testers = rw
```

如果你使用的是 SVNParentPath directive 来联网多个项目仓库，你可以使用 [repository:path] 的语法来引用一个具体的项目仓库。如果 administrators 用户组能够读写访问 *admin* 项目仓库下的文档，可使用以下的安全性定义：

```
[admin:/docs]
@administrators = rw
```

在前面的例子中能够访问 docs 目录的只有管理员们。事实上，admin 项目仓库能够被人访问到的只有 docs 目录，因为 Subversion 会禁止对所有目录的访问，除非被明确地许可可以这么做。经常会发现，把项目仓库的其它所有位置指定为只读访问挺有用的。这需要这么来写：

```
[/]
* = r
```

■ 使用钩子脚本来做访问控制

hook scripts

Subversion 可以配置数个“**钩子脚本**”，这些脚本在提交的特定时刻在服务器端运行。比如说，pre-commit 钩子在提交被接受之前运行，而且如果它返回了一个非零的退出状态，Subversion 会拒绝接受这次提交。

pre-commit 钩子不但知道哪些文件被改动了，还知道哪个用户在尝试改动它们。我们使用这些信息来设置基于路径的安全性——如果一个用户尝试

着改动一个没有权限的文件或者目录，我们的钩子脚本返回 1，这样 Subversion 就会停止提交。

Subversion 自带了 `commit-access-control.pl` 文件，它应该在你安装 Subversion 的时候被安装到了某个地方。比如说，对于 Fedora Core 5，它被放在`/usr/share/doc`，与 Subversion 文档在一起。

把 `commit-access-control.pl` 拷贝到你项目仓库的 `hooks` 目录。还要拷贝 `commit-access-control.cfg.example`，你应该给它改名，去掉`.example`扩展名。

在你的 hooks 目录中，重命名 `pre-commit.tmpl` 并且让它可执行：

```
hooks> mv pre-commit.tmpl pre-commit
hooks> chmod +x pre-commit
```

pre-commit 模版做两件事情——它检查日志信息是否包含了某些文本（你可以关心也可以不关心）然后运行访问控制脚本来检查用户的权限。

在 `commit-access-control.cf` 中配置权限，然后就可以用了。Python 爱好者可能会对 `svnperms.py` 感兴趣，它也包含在 Subversion 的分发包中，工作原理也是相似的。

访问文件不光是以只读模型读取的钩子脚本，你可能要为它们配置好系统上的安全设置。如果你使用 Apache 来伺服 Subversion 项目仓库，钩子脚本会以 Apache 用户的身份来运行，通常是 *httpd* 或者 *noboby*，而不是一个普通的用户。这可能意味着你的脚本不能执行修改文件或者发送邮件这样的操作，实际情况如何取决于你的配置。许多 Linux 分发版本包含了 SELinux 增强安全性，它使得特定的程序——比如 Apache 服务器——不能访问其常见的配置或者网站目录之外的文件。在这样的情况下，可能需要放松对钩子脚本的限制以令其能正常工作。一般可以通过检查安全日志的办法发现你的脚本是不是被 SELinux 或者别的安全设置给拦住了。

A.6 备份你的项目仓库

备份你的源代码意义重大，毕竟你的开发者都假设项目仓库是一个安全的地方，可以用来存储所有那些来之不易的劳动成果，Subversion 要么使用 Berkeley DB 要么使用“fsfs”文件系统来做你项目仓库的后台，而这些不能像普通文件一样备份。如果某人在备份过程中改动了项目仓库，项目仓库文件就可能会处于不一致的状态，导致备份无效。[6]

dumpfile

Subversion 提供了 `svnadmin dump` 命令来把项目仓库的内容抽取到一个可拿走的备份文件中。备份文件包含了项目仓库中的所有版本的信息，并且可以像普通文件一样备份。`svnadmin load` 命令取出备份文件的内容，并把它载入到一个项目仓库中去。这可以用来从一个备份中恢复原来的状态或者把项目仓库拷贝到另外一个位置。

■ 完整备份

取决于你的项目仓库有多大，以及你想要多久备份一次，或许能够一次完整地备份整个项目仓库。下面的命令创建了项目仓库的一个完整备份：

```
mike> svnadmin dump ~/svn-repos > dumpfile.041113
* Dumped revision 0.
* Dumped revision 1.
   :      :      :
* Dumped revision 48.
```

Subversion 把项目仓库的所有版本打印到了控制台上，然后我们把这些内容存储到 `dumpfile.041113` 中。最终的文件包含了项目仓库中的所有内容。一个典型的备份文件是高度可压缩的，然后就可以拿去做备份了。

使用 `svnadmin` 创建的项目仓库快照不会出现内部状态不一致的情况，即便在备份的过程中有改动提交到了服务器也没有关系。这意味着你在运行 `svnadmin dump` 时无须停止对项目仓库的访问。

---

[6] 使用“fsfs”项目仓库，你更可能得到的是一致的备份，特别是你备份文件的时候有一定的顺序。参见 `http://svn.collab.net/repos/svn/trunk/notes/fsfs`，那里有更具体的说明。

要从备份文件把老的项目仓库恢复到一个新的项目仓库时，可使用svnadmin load。首先我们创建一个新的项目仓库（老的那个可能还没有完全地坏掉，所以不要删掉了），然后载入备份文件：

```
mike> svnadmin create svn-repos2
mike> svnadmin load svn-repos2 < dumpfile.041113
<<< Started new transaction, based on original revision 1
     * adding path : sesame ... done.
     * adding path : sesame/trunk ... done.
     * adding path : sesame/trunk/Day.txt ... done.
     * adding path : sesame/trunk/Number.txt ... done.
------- Committed revision 1 >>>
   :       :      :
<<< Started new transaction, based on original revision 48
  * adding path : sesame/trunk/common/HibernateHelper.java
    COPIED... done.
  * adding path : sesame/trunk/contacts/Contacts.hbm.xml
    COPIED... done.
  * editing path : sesame/trunk/contacts/Contacts.java ... done.
------- Committed revision 48 >>>
```

Subversion 从备份文件中回放了每个版本，并且把改动提交给项目仓库。一旦载入过程结束，项目仓库就可以用了，其内容应该和备份创建的时候是一样的。

■ 增量备份

每天都做完整备份的话，很快磁盘空间就会被吃光的。幸运的是，可用svnadmin dump接受一个--incremental选项，以及--revision选择一个版本范围来产生尺寸小一些的备份文件。

比方说，你已经有了一个包含了1到100版本的备份文件，但是项目仓库现在的版本是104。你可以通过运行以下命令创建一个增量的备份文件：

```
work> svnadmin dump --incremental --revision 100:104 \
                    /home/svn-repos
```

结合每周备份，每日增量的方式可以让你有安全感，同时不需要耗费惊人的磁盘空间。只要写一点脚本，你就能建立一个每周和每日备份的例行程序，它能记住每次需要备份的版本号。

下面的程序是每周备份的脚本。虽然它看起来不过是一个非常简单的Perl脚本，不过它演示了如何使用svnlook youngest来找出你项目仓库当前所在的版本号。

svn-backup/weekly-backup.pl

```
#!/usr/bin/perl -w
#
# Perform a weekly backup of a Subversion repository,
# logging the most-recently-backed-up revision so an
# incremental script can be run other days.
$svn repos = "/home/mike/svn-repos";
$backups dir = "/home/mike/svn-backup";
$next backup file = "weekly-full-backup." . 'date +%Y%m%d';

$youngest = 'svnlook youngest $svn repos';
chomp $youngest;

print "Backing up to revision $youngest\n";
$svnadmin cmd = "svnadmin dump --revision 0:$youngest " .
                "$svn repos > $backups dir/$next backup file";
'$svnadmin cmd';
print "Compressing dump file...\n";
print 'gzip -9 $backups dir/$next backup file';
open(LOG, ">$backups dir/last backed up");
print LOG $youngest;
close LOG;
```

运行这个脚本会把你的项目仓库备份到一个叫做 weekly-full-backup.yyyymmdd 的文件中，并且使用 gzip 进行压缩。它还把最新的版本备份到一个叫做 last_backed_up 的文件中：

```
svn-backup> ./weekly-backup.pl
Backing up to revision 638
* Dumped revision 0.
* Dumped revision 1.
* Dumped revision 2.
    .   .   .
* Dumped revision 638.
Compressing dump file...
```

每天备份的脚本使用每周备份的脚本所保存的版本号来把刚改动的内容备份出来，而不是一次备份整个项目仓库：

```
#!/usr/bin/perl -w
#
# Perform a daily backup of a Subversion repository,
# using the previous most-recently-backed-up revision
# to create just an incremental dump.
$svn repos = "/home/mike/svn-repos";
$backups dir = "/home/mike/svn-backup";
$next backup file = "daily-incremental-backup." . 'date +%Y%m%d';

open(IN, "$backups dir/last backed up");
$previous_youngest = <IN>;
chomp $previous_youngest;
close IN;

$youngest = 'svnlook youngest $svn repos';
chomp $youngest;
if($youngest eq $previous youngest) {
    print "No new revisions to back up.\n";
```

```
    exit 0;
}
# We need to backup from the last backed up revision
# to the latest (youngest) revision in the repository
$first_rev = $previous_youngest + 1;
$last_rev = $youngest;

print "Backing up revisions $first_rev to $last_rev...\n";
$svnadmin_cmd = "svnadmin dump --incremental " .
                "--revision $first_rev:$last_rev " .
                "$svn_repos > $backups_dir/$next_backup_file";
 `$svnadmin_cmd`;
print "Compressing dump file...\n";
print `gzip -9 $backups_dir/$next_backup_file`;
open(LOG, ">$backups_dir/last_backed_up");
print LOG $last rev;
close LOG;
```

svn-backup/daily-backup.pl

在改动了一些东西之后，运行这个脚本只会把新添加的版本给备份出来：

```
svn-backup> ./daily-backup.pl
Backing up revisions 639:641
* Dumped revision 639.
* Dumped revision 640.
* Dumped revision 641.
Compressing dump file...
```

每日的增量备份比完整备份要小得多，但是因为没有包含所有足够的信息，所以在灾难袭来的时候没法用它来恢复项目仓库。要做恢复的话，你要首先载入你最近的完整备份，然后载入每天的备份：

```
svn-backup> mkdir newrepos
svn-backup> svnadmin create newrepos
svn-backup> zcat weekly-full-backup.20041129.gz | \
                  svnadmin load newrepos
<<< Started new transaction, based on original revision 1
    * adding path : branches ... done.
    * adding path : tags ... done.
    * adding path : trunk ... done.
          :    :    :    :
svn-backup> zcat daily-incremental-backup.20041130.gz | \
                  svnadmin load newrepos
<<< Started new transaction, based on original revision 639
     * editing path : trunk/ccnet/lib/NetReflector.dll ... done.
------- Committed new rev 639 (loaded from original rev 639) >>>
          :    :    :    :
```

PDG

附录 B

迁移到 Subversion

Migrating to Subversion

现在你应该相信使用 Subversion 好处多多了。你会喜欢它对基本单位的支持，对把修改打包在一块的支持，对高速网络协议的支持，对真正的分支和合并的支持。你甚至说服了你的老板，让他相信了 Subversion 正是团队所需要的。离伟大的胜利只剩最后一个小问题了，那就是半打的项目加上数年的历史记录还存在于你现有的版本控制工具之中呢。

幸运的是，Subversion 的开发者想到了人们可能要把历史信息保存起来，并且提供了工具把现有的 CVS[1]项目仓库迁移到 Subversion 来。你还能找到第三方的工具把它们从 ClearCase、Perfoce 甚至 Visual SourceSafe 迁移过来。

在迫不及待地阅读本章之前，值得先思考一下：转换你现有的项目仓库是不是一个合理的迁移策略。在使用了 Subversion 之后，如果你并不是很在乎之前的历史信息，只要把你的源代码从原来的版本控制工具中导出，然后再导入到 Subversion 中就行了。如果你确实想要让历史信息留在那儿的话，请把旧的项目仓库变成只读的，尽管不在新的 Subversion 项目仓库中。

在本章中，假定你想要转换所有的历史并且我们要做的迁移是从 CVS 到 Subversion。

[1] `cvs2svn` 还能转换 RCS 项目仓库到 Subversion，因为 RCS 是 CVS 使用的底层格式。

B.1 获得 cvs2svn

cvs2svn 项目和 Subversion 主项目是放在同一个地方的，都伺服在 Tigris 上。[2]下载对应于你所使用的 Subversion 版本的包。在写作本书的时候，cvs2svn 只对 Subversion 1.2 或者 1.3 可用，更老的版本就不再支持了。

cvs2svn 是一个 Python 脚本，设计在 Unix 下运行。你可能想要让它在 Windows 下也能用，可以使用 Cygwin[3]这个 Linux 模拟工具，我们推荐使用真正的 Unix 主机来做这件事情。你可能还需要安装 rcs，因为 cvs2svn 使用它来访问你 CVS 项目仓库的内容（你可以通过安装 CVS 达到这个目的，但是使用“真正”的 RCS 更保险一点）。

你可以把 cvs2svn 安装在你的系统上，让每个人都能看到它或者只是把它运行为一个普通的用户。两种方式都可行，但是如果你不是 Python 大牛的话，安装成系统级别的更容易一些。登录成 root，运行以下命令：

```
tmp> tar -xzf cvs2svn-1.2.1.tar.gz
tmp> cd cvs2svn-1.2.1
cvs2svn-1.2.1> make install
./setup.py install
running install
     :     :     :
copying build/lib/cvs2svn_rcsparse/compat.py -> ...
copying build/lib/cvs2svn_rcsparse/debug.py -> ...
     :     :     :
byte-compiling /usr/lib/python2.3/.../default.py to default.
pyc
byte-compiling /usr/lib/python2.3/.../texttools.py to text
tools.pyc running install_scripts
copying build/scripts-2.3/cvs2svn -> /usr/bin
changing mode of /usr/bin/cvs2svn to 775
```

B.2 选择转换多少内容

cvs2svn 会对你现有的 CVS 项目仓库做一个彻底的转换，包括所有的分支和标签。它还分析历史，查找同时改动的，日志消息相同的文件，通过这种方式把 CVS 的每个文件的版本历史转换为 Subversion 的改动集的方式。所有这些都是费时的工作，费时多少取决于你 CVS 项目仓库的大小。

---

[2] http://cvs2svn.tigris.org/
[3] http://www.cygwin.com

如果你不想要转换所有的历史，就可以不用这么做。指定哪些分支是你感兴趣的，这样可以节省转换的时间并且只占用新的 Subversion 项目更少的地方。cvs2svn 接受大量的命令行参数，但是最有用的要数--exclue 了，它设置一个正则表达式来匹配你想要在转换中跳过的标签和分支。

值得注意的是，cvs2svn 是设计用来一次性从 CVS 到 Subversion 的转换的；它不能以增量的方式同步两个系统。

B.3 转换你的项目仓库

让我们假设你想完整地转换你 CVS 项目仓库中的所有东西。第一步是确保所有人都把改动签入到了 CVS 中了，并且他们知道你要开始做这次转换。接下来把你的 CVS 项目仓库离线，从而不会再有新的改动提交进去。最后一步准备工作，也是最重要的一步，是拷贝一份 CVS 项目仓库。你需要拷贝整个 CVSROOT，因为 cvs2svn 要对它进行转换。我再说一遍：**拷贝一份 CVS 项目仓库，用这份拷贝去做转换。**

cvs2svn 通过创建 Subversion 的备份文件来工作，和 svnadmin dump 创建的文件是一个格式。备份文件在转换后用 svnadmin load 载入到 Subversion 项目仓库中。你可以使用 cvs2svn 的-s 选项来指定一个你想要创建新 Subversion 项目仓库的目录，快捷地完成这个过程。

这儿我们将要转换一个放在 SourceForge 上的叫 Testsweet 的项目。SourceForge 上的项目有一个很好的地方是你可以用 CVS 每日快照的形式下载一个项目仓库的压缩拷贝，这正是 cvs2svn 需要转换的文件。[4]如果你想要试看 cvs2svn，但是不想拿你自己的 CVS 项目仓库来试的话，没准这正是你想要的。

[4] Testsweet 的每日 CVS 快照位于 http://cvs.sourceforge.net/cvstarballs/testsweet-cvsroot.tar.bz2

拷贝你的 CVS 项目仓库到一个临时目录。在这个例子中，我们把 Testsweet 的 CVS 项目仓库拷贝到一个本地叫 testsweet 的目录中，而且我们要把它转换成一个在 testsweet-repos 的 Subversion 项目仓库。cvs2svn 在转换过程中会替我们创建项目仓库的目录并初始化好它：

```
tmp> cvs2svn -v -s testsweet-repos testsweet
----- pass 1 -----
Examining all CVS ',v' files...
testsweet/CVSROOT/checkoutlist,v
testsweet/CVSROOT/commitinfo,v
testsweet/CVSROOT/config,v
    :       :       :
```

我们要求冗余信息也被打印出来（-v 选项），所以 cvs2svn 在转换过程中产生了大量的输出。Testsweet 是一个很小的项目，只要几秒钟就转换好了。当它完成时，cvs2svn 打印出一些统计信息：

```
cvs2svn Statistics:
------------------
Total CVS Files:              161
Total CVS Revisions:          218
Total Unique Tags:              1
Total Unique Branches:          0
CVS Repos Size in KB:        2716
Total SVN Commits:              10
First Revision Date:  Fri Nov 21 18:12:21 2003
Last Revision Date:   Thu Jun 24 12:35:29 2004
```

我们现在可以使用 svn ls 来浏览新的项目仓库，可以看到我们有了常见的 trunk，tags 和 branches 目录：

```
tmp> svn ls file:///tmp/testsweet-repos
branches/
tags/
trunk/
tmp> svn ls file:///tmp/testsweet-repos/trunk
CVSROOT/
testsweet/
```

现在把 Testsweet 项目放在项目仓库的根目录中，这可能不是我们想要的。cvs2svn 可以让我们指定 trunk，tags 和 branches 目录的位置：

```
tmp> cvs2svn --trunk=testsweet/trunk        \
             --branches=testsweet/branches  \
             --tags=testsweet/tags          \
             -s testsweet-repos testsweet
```

你新创建的项目仓库立马就可以拿来用了。只须用 svnserve 或者 Apache 联好网络，签出工作拷贝，就可继续写你的代码！

附录 C

第三方的 Subversion 工具

Third-Party Subversion Tools

Subversion 是以一套命令行程序的形式发布的——svn，svnadmin，svnserve，等等。命令行是很容易使用的，但是很多人还是喜欢使用一些界面更友好的工具。幸运的是，Subversion 提供了丰富的 API 供第三方开发人员使用，因此他们可以编写附加的客户端和工具。

C.1 TortoiseSVN

Tortoise 是直接与 Windows 文件浏览器集成的 Subversion 前端。安装好了之后，你只需要浏览你的电脑就可以看到你文件和目录的状态。Tortoise 在未修改的文件旁边标记一个绿色的勾，在你修改过的文件旁边标记一个红色的小警告标志。Tortoise 还给解决冲突，管理标签和创建分支等任务提供了方便的自动化帮助。

在本节中，我们将使用 Tortoise 执行一些我们前面用命令做过的日常工作。

■ 下载和安装

下载 TortoiseSVN，[1]并且运行适合你的 Windows 版本的安装程序，之后应该弹出一个欢迎窗口。选择要安装到的路径（或者接受默认的目录 C:\Program Files\TortoiseSVN），接着选择是否要给计算机上的所有用

[1] TortoiseSVN 可以在 http://tortoisesvn.tigris.org/找到。

户安装 Tortoise 或者只是安装给你自己一个人用。需要你来决定的就这些了；其余的任务，安装程序都会自动完成。

Tortoise 安装程序会要求你重启计算机。不像大部分的安装程序那样，重启不重启都没多大关系，Tortoise 确实是在乎这点的；因为它要与 Windows 文件浏览器集成，所以必须重启才能注册自己。

■ 签出

打开文件浏览器窗口，切换到你想要签出工作拷贝的目录。这儿，我们使用 `C:\work`。在目录上单击鼠标右键，会出现一个包含了 Tortoise 的 Subversion 集成的菜单，如图 C.1 所示。

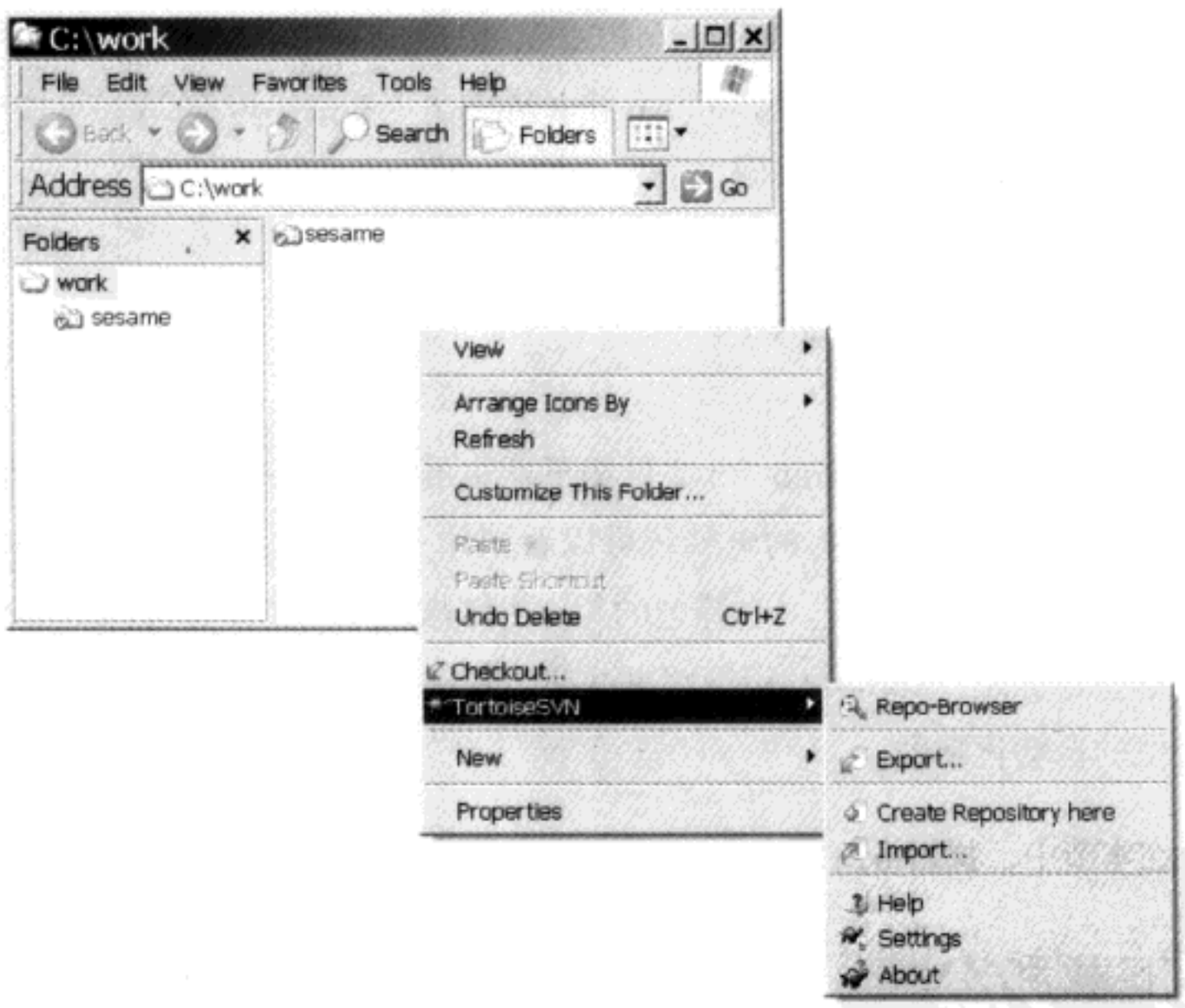


图 C.1 Tortoise 的上下文菜单

在上下文菜单中选择 *Checkout*...，它会打开一个对话框询问你想要签出什么。使用拿来做试验的项目仓库 `file:///C:/svn-repos/sesame/trunk`，并签出到 `C:\work\pincess`（`sesame` 已经是工作拷贝了，所以我们新建一个 Princess）。

Tortoise 会在签出 Sesame 项目的过程中闪动进度条，最后会留下一个新的 `princess` 目录。进入目录，你会看到 `Day.txt`，`Number.txt` 以及 Sesame 项目的其余文件。因为所有的文件都是最新的，Tortoise 以绿色的勾标记它们，如图 C.2 所示。

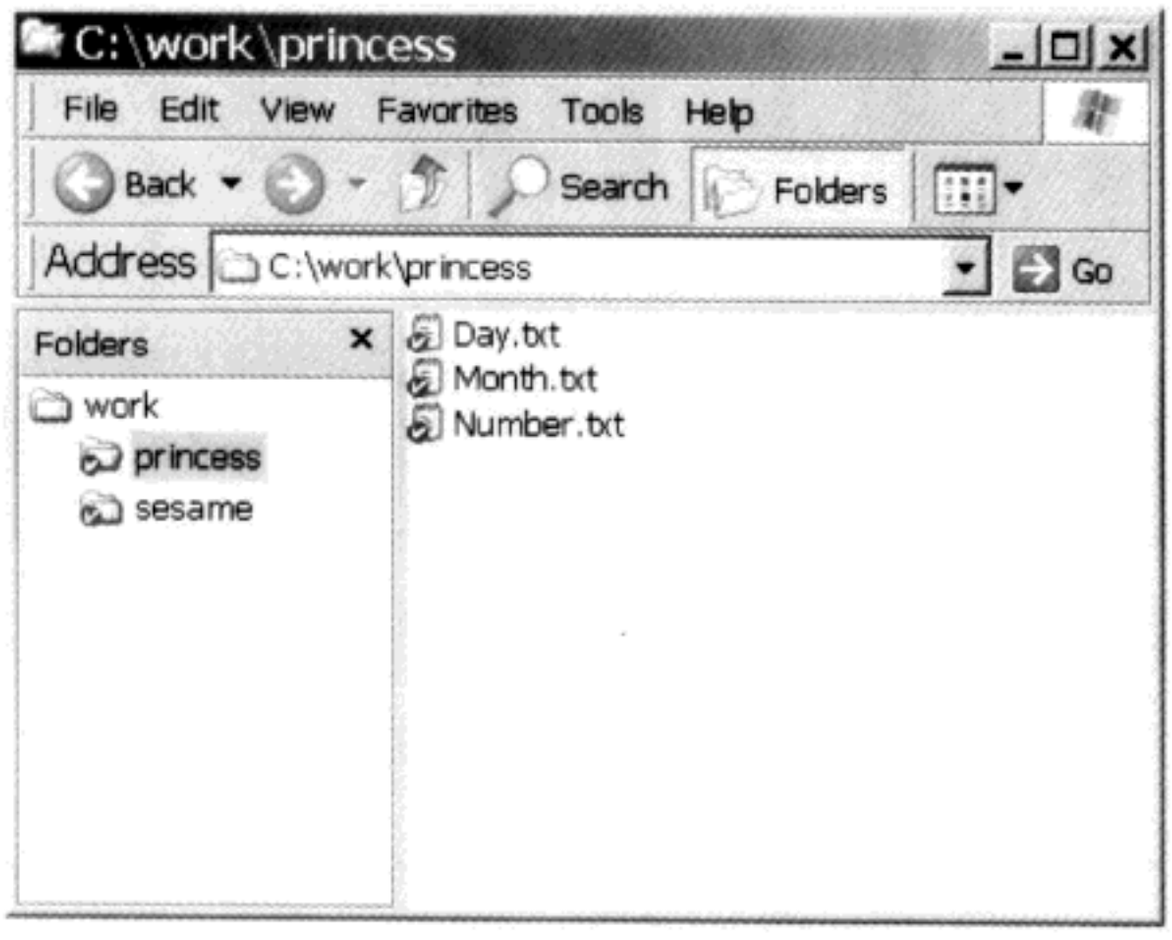


图 C.2 新签出的 Sesame 项目

■ 做改动

对 `Number.txt` 做一些改动，比如说让 *three* 大写。在你保存文件之后，Tortoise 会用红色的惊叹号标记它。[2] 父目录 princess 也会被标红。在 `Number.txt` 上点击鼠标右键，选择 *TortoiseSVN > Diff*。

[2] 你可能需要按 F5 键让 Windows 刷新屏幕，显示新的图标。

这会出现一个类似于图 C.3 的 TortoiseMerge 窗口显示你做过的改动。

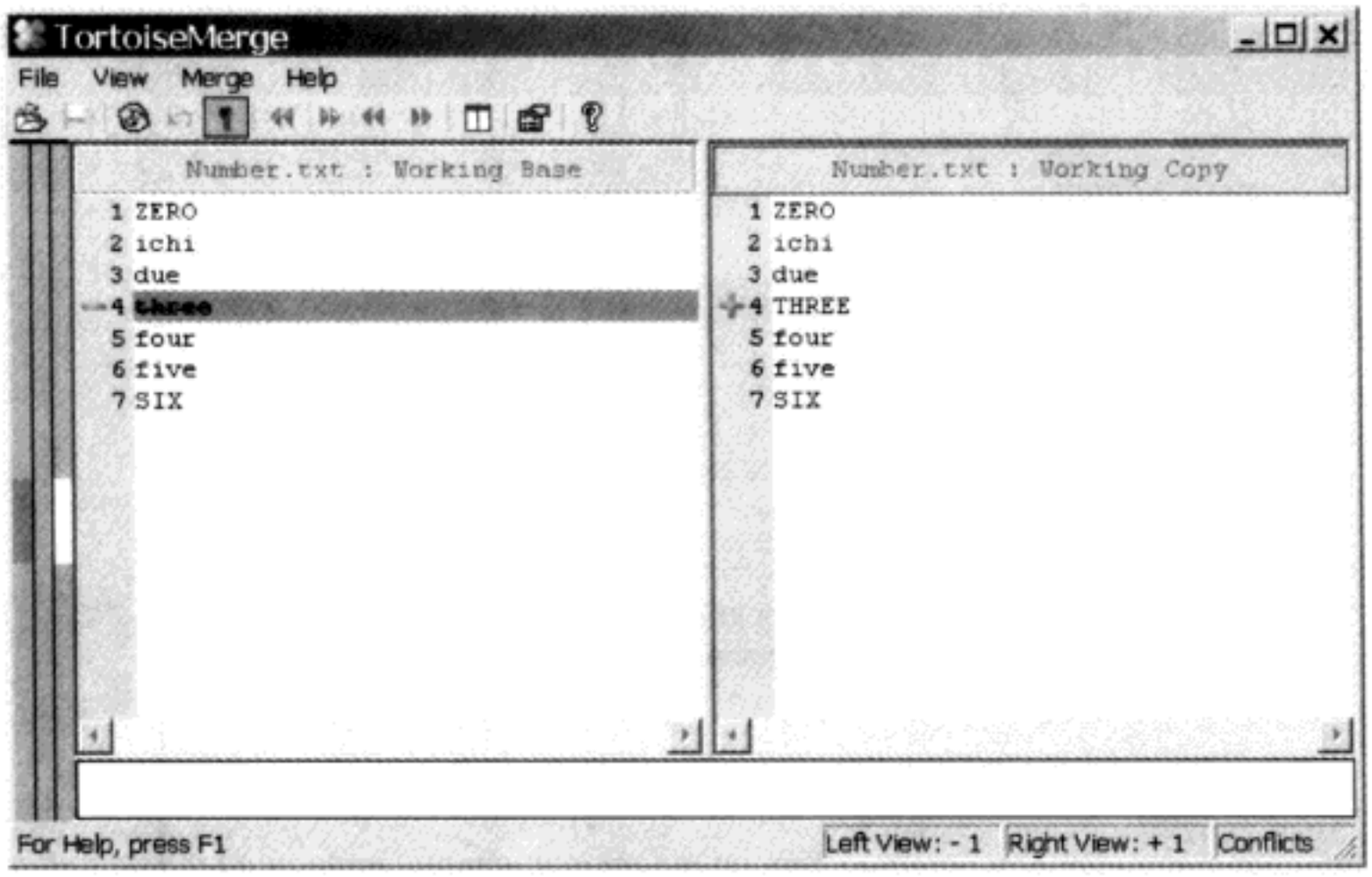


图 C.3 查看你的改动

添加一个新文件同样是很直观的。创建一个叫 `Year.txt` 的新文件，并且保存在你的工作拷贝中。Subversion 对于这个文件还一无所知，所以 Tortoise 不会标记它。在新文件上单击鼠标右键，选择 *TortoiseSVN > Add*。Tortoise 会打开一个窗口询问你是不是确认要添加这个文件，这在你要一次添加很多文件的时候会比较管用。点击 OK 按钮，Tortoise 就会添加这些文件。因为 Subversion 现在知道了 Year.txt，所以它显示了一个蓝色的“加号”图标。

■ 签入

在 `princess` 目录上点击鼠标右键，并且选择 *Commit...*。Tortoise 会显示所有你改动过的文件并且提示要你输入提交消息，如图 C.4 所示。此时，你通过取消对某些文件的选择决定不提交它们。如果你不确定你改动了什么，双击文件会弹出一个 diff 窗口，你可以从中查看你的改动。

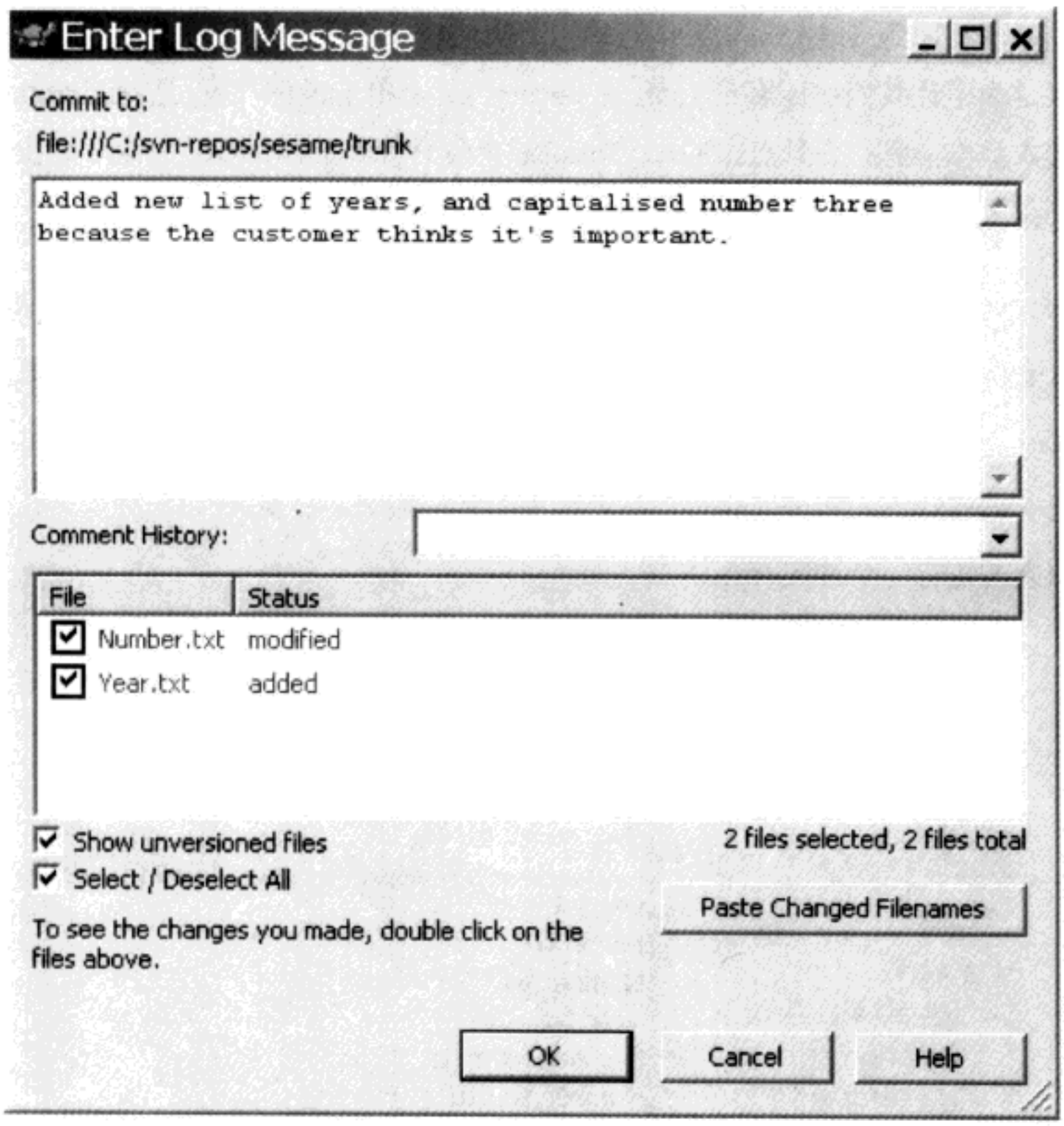


图 C.4 TortoiseSVN 提交窗口

输入一条提交消息描述你的改动（更重要的是，你为什么做这些改动）并且点击 OK。Tortoise 会在提交你的改动到项目仓库的过程中，不断更新窗口。

■ 解决冲突

作为光速穿越 Tortoise 旅程的一部分，看看在遇到冲突时它能如何帮助我们吧。签出另外一份 Sesame 项目的拷贝，这次是签出到 `C:\work\aladdin`。我们使用这个目录来模拟团队中另外一个人，Aladdin 的动作。编辑 `Number.txt`，把 *five* 改为 *cinco*，然后提交改动。

现在回到 princess 工作拷贝，编辑同一个文件，把 *five* 改为 *cinq*，这次是为了让我们法国客户满意。在 princess 上单击鼠标右键，选择 Commit，输入日志消息，并且点击 OK。Tortoise 会告诉你 Number.txt 过期了，提交失败了。它还会建议你更新你的工作拷贝后再提交。

遵循 Tortoise 的建议，点击鼠标右键选择 *Update*，更新，Tortoise 从项目仓库获得最新版本时更新窗口会弹出。取决于你机器的速度，在更新 Number.txt 的时候，你可能会注意到一条红线，表示一个冲突。Tortoise 会把 Number.txt 和 princess 目录都用警告三角表示，如图 C.5 所示，让你知道这儿有一个冲突存在。

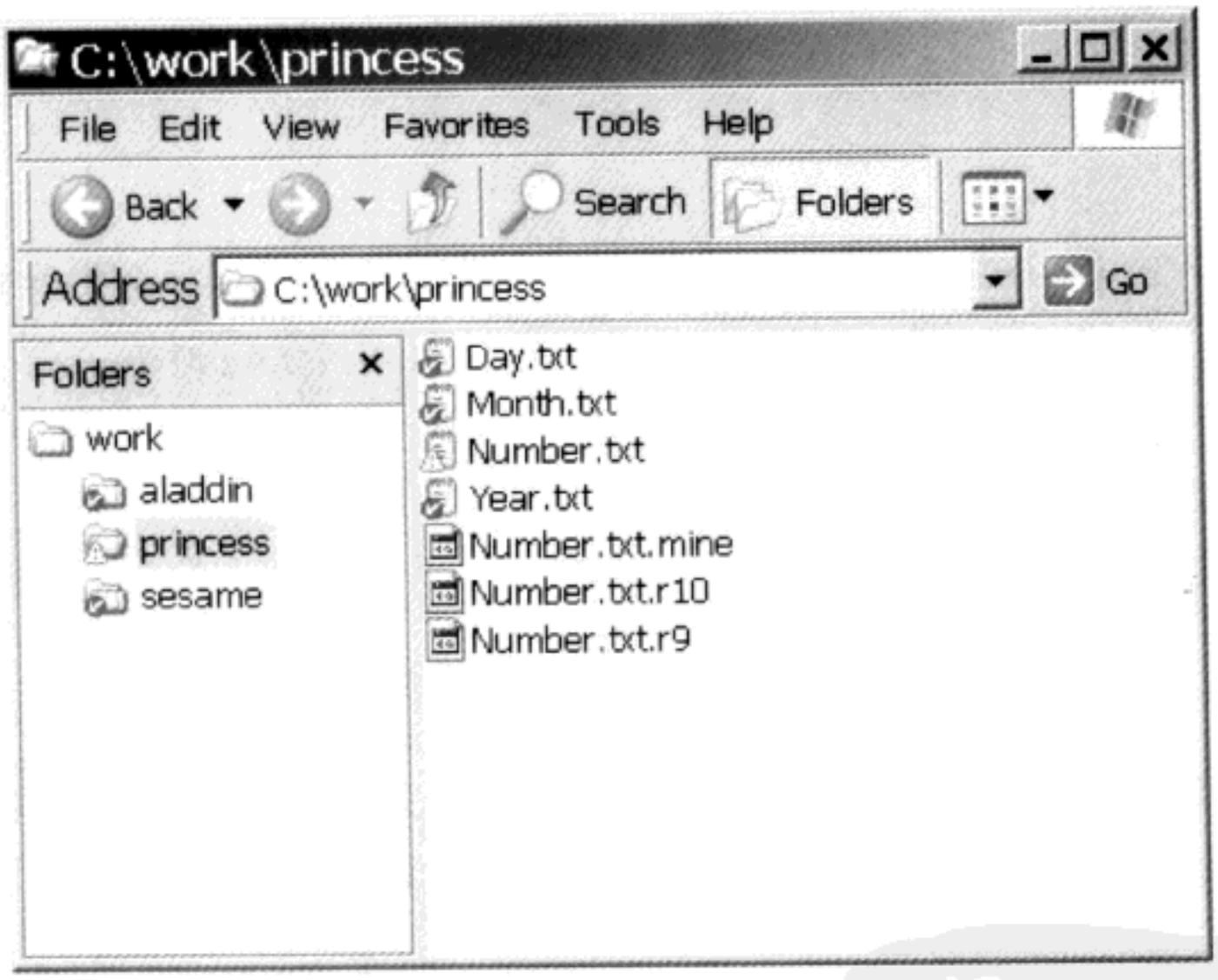


图 C.5 冲突中的 Number.txt

Tortoise 还额外地保存一份 Number.txt 以帮助你解决冲突。可以注意到还有.mine，.r9 和.r10。首先，.mine 是你的版本，包括了你修改的内

容。其次，.r9 包含了你的改动所基于的版本——这是 Princess 开始编辑它时的那个版本。最后，r10 包含了与你的改动相冲突的版本。这些改动是 Aladdin 提交的。

幸运的是，Tortoise 提供了三岔[3]合并工具来帮助人们更容易地修正冲突。在 Number.txt 文件上点击鼠标右键，并且选择 *TortoiseSVN > Edit Conflicts*。Tortoise 会弹出如图 C.6 那样的合并窗口。

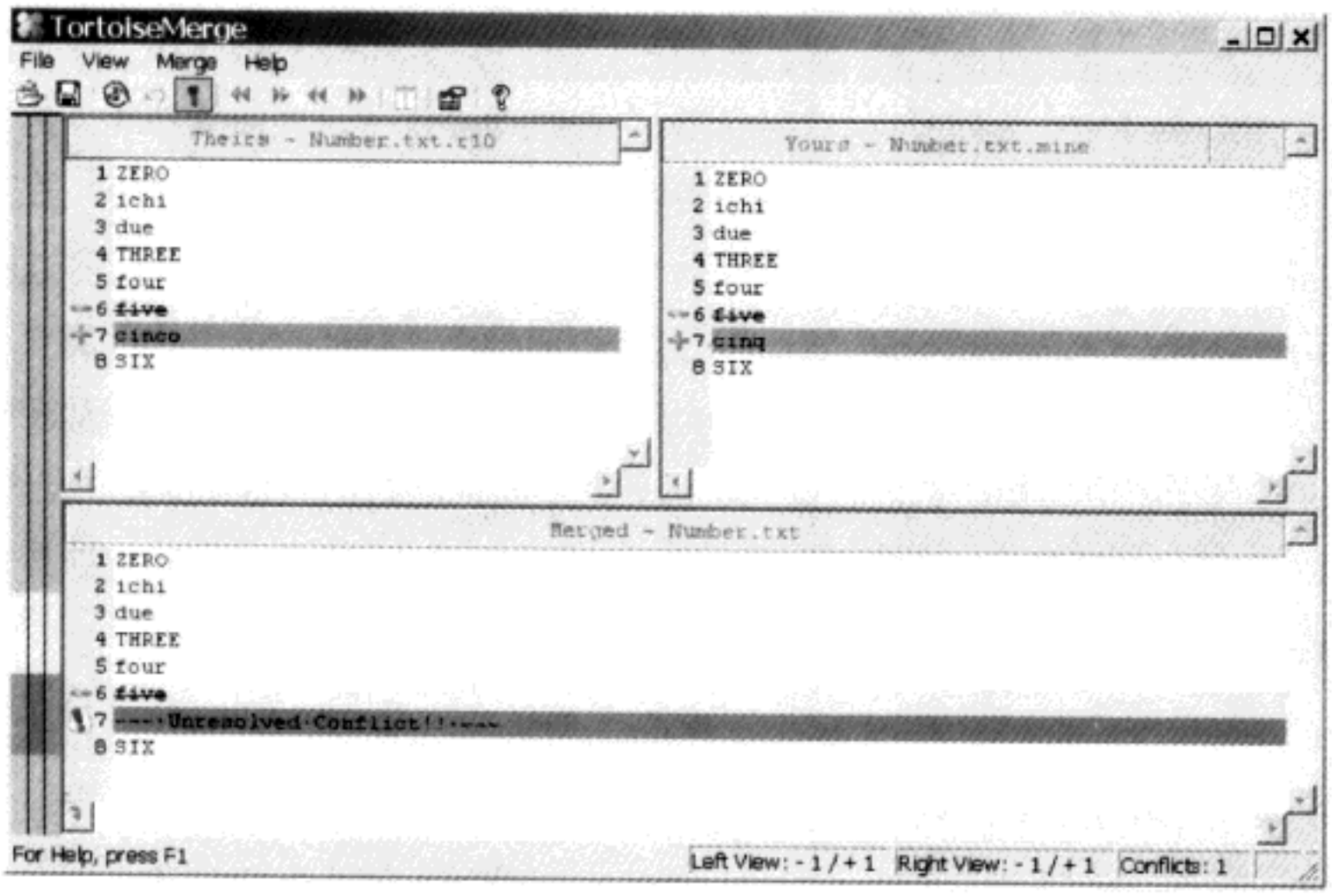


图 C.6 Tortoise 合并窗口

TortoiseMerge 同时显示两份改动，以及在窗口的下半部分显示合并的结果。在本例中，我们非常确信 cinq 是正确的，所以选择保留我们的改动（保留 Princess 的改动而不是 Aladdin 的）。在 cinq 上点击鼠标右键，并且选择 *Use this text block*。Tortoise 把更新结果更新在窗口的下半部分显示结果的区域。如果你有多于一处的冲突，可以在两份改动之间选择使用哪份，直到你

[3] 之所以叫三岔合并这名字是因为它合并了文件的原始版本和两个人的改动。

满意了为止。现在关闭合并窗口。Tortoise 会询问你是否想要保存你的改动；因为你对合并的结果已经满意了，所以回答“yes”，然后 Tortoise 会把合并窗口给关掉。

你会注意到 Tortoise 仍然用小的警告三角标记着文件。现在我们已经解决了冲突，我们需要告诉 Tortoise，一切都正常了。在 `Number.txt` 上单击鼠标右键，选择 *TortoiseSVN > Resolved*。Tortoise 会清理掉 `.mine`，`.r9` 和 `.r10` 这些文件并且以惊叹号标记 `Number.txt`，表示你修改过它。现在就可以和往常一样完成签入过程了。

TortoiseSVN 为做分支、打标签和合并改动提供了一些便捷功能。但是在这里没有细致地讲解每一项功能，我建议你每项都拿来试着用一下。在这里再蜻蜓点水地说说那极其好用的项目仓库浏览器，你可以通过选择 *TortoiseSVN > Repo-Browser* 打开它。这个聪明的小工具让你可以在不需要工作拷贝的情况下到项目仓库中四处看看。如果你想要找出项目的分支都在哪儿、以及第三方的源代码放在哪儿了诸如此类的问题的话，使用它会很方便。图 C.7 显示了浏览中的位于 `http://svn.collab.net` 的 Subversion 项目仓库。

C.2 IDE 集成

Subversion 的 IDE 集成在 Subversion 1.0 之后迅速地成熟起来，许多流行的 IDE 现在都有了官方的支持。只使用命令行或者 Tortoise 是可能的，但是编辑器紧密集成了版本控制工具的话，大部分用户会更习惯一些，所以如果可以的话去，建议你去调查一下你使用的编辑器是不是有 Subversion 的支持。

Eclipose，流行的开源 Java IDE，有一个叫做 Subclipse 的插件来集成 Subversion，可以从 `http://subclipse.tigris.org/` 下载到。

IntelliJ IDEA，另外一个流行的 Java IDE，自从版本 5.0 之后就对 Subversion 有了完整的支持。IDEA 可以从 `http://www.jetbrains.com/` 下载到。

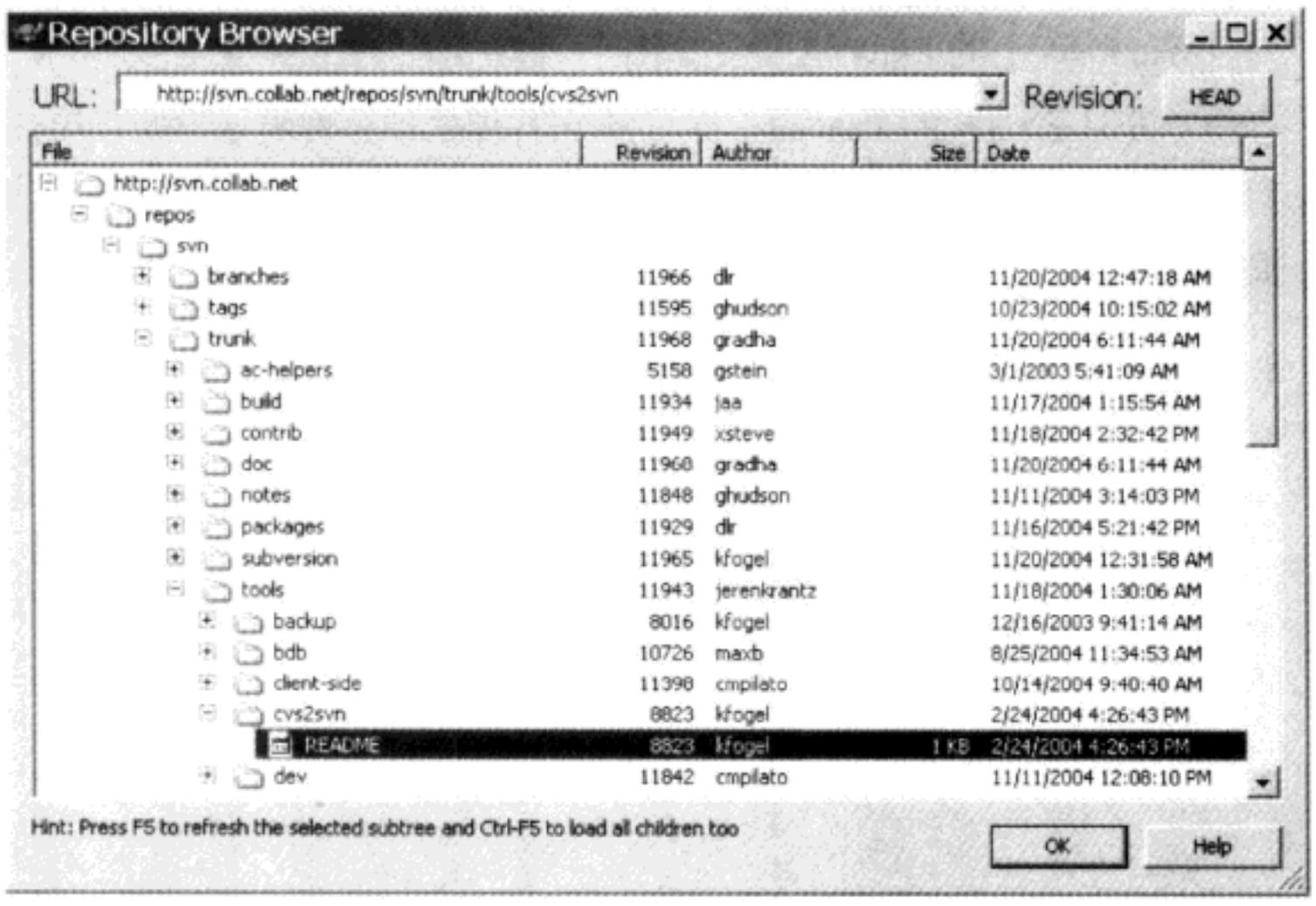


图 C.7 运行中的 Tortoise Repo-Browser

Ankhsvn 提供了与 Visual Studio 的集成，从 `http://ankhsvn.tigris.org/` 可以下载到。注意如果你使用的是 Visual Studio 的 web 项目，它们可能与 Subversion 的 `.svn` 管理目录不兼容。TortoiseSVN 有一个特殊的“directory hack”的选项，会使用 `_svn` 作为替代的目录名。记住这样创建的工作拷贝与普通的工作拷贝是不兼容的，所以你可能需要在启用了这个 hack 之后重新签出一遍。

C.3 其他工具

在开发者把改动签入到项目仓库时，SVN::Notify 能发送一封彩色的 HTML 邮件。对于你的团队来说，这也许是一个极好的交流工具。SVN::Notify 从 CPAN 可以下载到：`http://search.cpan.org/dist/SVN-Notify/`。

如果你在 Mac 上使用 XCode，你或许需要一个密钥管理器来让 SSH 连接可以正常工作。SSHKeychain 提供了“Mac OS X 的无痛密钥管理”，可以从 `http://www.sshkeychain.org/` 下载到。

Windows 上的 Putty 这一套 SSH 客户端工具[4]，用来连接 svn+ssh 也很好用。并且它还包含 Pageant，一个密钥管理代理。如果你想要避免在你访问 svn+ssh 的项目仓库时都要输入密码的话，Putty 手册的第 8 章第 9 章[5]也值得一读。

[4] http://www.chiark.greenend.org.uk/~sgtatham/putty/
[5] http://the.earth.li/~sgtatham/putty/0.56/htmldoc/Chapter9.html

附录 D

高级话题

Advanced Topics

D.1 编程访问 Subversion

Subversion 支持大量的**语言绑定**，让你可以通过你最喜欢的语言访问 Subversion 的 API。语言绑定使用 *Swig*（Simplified Wrapper and Interface Generator，简化封装和接口产生器），基本上就是把 C 的 API 暴露给其它语言。Swig 绑定有 C，C++，C#，Java，Perl，Python 和 Ruby 的版本。在这些当中，Python 绑定可能是最流行的也绝对是最成熟的。希望这些绑定随着时间的推移，能够越来越成熟。

在不同的操作系统上，安装语言绑定的过程各不相同，而且 Unix 上支持得似乎要更好一些。对于 Fedora Core 5，安装 Perl 绑定简单到只要 `yum install subversion-perl`。

为了开发一个使用 C 的 API 封装之外的选择，一群开发人员决定用纯 Java 实现 Subversion 的客户端。当他们开始这么做的时候，来自 Subversion 开发人员的意见是“干嘛要这么麻烦呢，用 Swig 绑定好了”但是他们坚持下来了，而且做出了一个很棒的实现。JavaSVN 库被 JetBrains 用来作为它们的 IDE 与 Subversion 集成的基础，这个实现正是他们所想要的。

TMate JavaSVN 库可以从 `http://tmate.org/svn/`下载到，而且我们在下节中会使用它作为例子。

■ 一个简单的 Subversion 客户端

从 TMate 网站下载独立版本的 JavaSVN，把它解压缩到你的硬盘上。你可以找到开发用的 `jar` 文件，以及一个 `doc` 子目录，其中包含了库的完整 JavaDoc 文档。要在项目中使用 JavaSVN，只要把 `javasvn.jar` 包含到你的 classpath 中就行了。

JavaSVN 库包含了各种各样的函数，用来与 Subversion 项目仓库通信以及操作本地的工作拷贝。作为第一个例子，我们将连接到 CollabNet 的 Subversion 项目仓库并且打印出一个目录列表。

远程操作是通过 `SVNRepository` 的实例进行的。因为我们使用 `http` 连接方式访问项目仓库，所以必须在使用它之前初始化 DAV 项目仓库工厂：

```
String reposUrl = "http://svn.collab.net/repos/svn";
SVNURL url = SVNURL.parseURIEncoded(reposUrl);

DAVRepositoryFactory.setup();
SVNRepository repository = SVNRepositoryFactory.create(url);
```

现在我们有了一个项目仓库实例，可以对远程项目仓库执行操作了。例如，可以使用 `getDir()` 方法获取目录列表：

```
Map<String, String> dirProps = new HashMap<String, String>();
List<SVNDirEntry> dirEntries = new ArrayList<SVNDirEntry>();
repository.getDir("/trunk/subversion",      -1,      dirProps,
dirEntries);
```

我们提供了一个项目仓库中的路径，`/trunk/subversion`，以及一个版本号，在本例中是-1，这表示我们感兴趣的是最新版本。我们还提供了一个空的 map，JavaSVN 会把被列目录的属性放在那儿，而把列出来的条目放在提供的空的 list 中。

每个 `SVNDirEntry` 代表被列目录中的一个文件或者目录。条目有大量属性，诸如名字、尺寸、版本、创建日期、最后被改动的日期，等等。这儿有一个列出 CollabNet Subversion 项目仓库中的文件的例子程序：

```
import org.tmatesoft.svn.core.*;
import org.tmatesoft.svn.core.io.*;
import org.tmatesoft.svn.core.internal.io.dav.*;
import java.util.*;
```

```
public class ListDirectory
{
  public static void main(String[] args)
         throws SVNException
{
    String reposUrl = "http://svn.collab.net/repos/svn";
    SVNURL url = SVNURL.parseURIEncoded(reposUrl);

    DAVRepositoryFactory.setup();
    SVNRepository repository = SVNRepositoryFactory.create(url);

    Map<String, String> dirProps = new HashMap<String, String>();
    List<SVNDirEntry> dirEntries = new ArrayList<SVNDirEntry>();
    repository.getDir("/trunk/subversion",    -1,    dirProps,
    dirEntries);
    for (SVNDirEntry dirEntry : dirEntries) {
      printEntry(dirEntry);
    }
}
private static void printEntry(SVNDirEntry entry) {
  if(entry.getKind() == SVNNodeKind.DIR) {
     System.out.println("Directory: " + entry.getName());
    } else {
    System.out.println("File: " + entry.getName()
            + ", size " + entry.getSize()
            + ", last modified by " + entry.getAuthor());
    }
  }
}
```

java/ListDirectory.java

■ 监视项目仓库的改动

Subversion 的钩子脚本让你可以截获感兴趣的事件（比如改动被提交了），但是你可能不是总想用钩子脚本的。在某些情况下，你可能不能使用钩子，因为项目仓库的安全问题或者访问受到了限制。许多持续集成工具使用一种“轮询”的策略来检查项目仓库中是不是有东西被改动过了，如果有的话，自动作一些有用的事情，比如构建最新的代码并运行测试。JavaSVN 库可以用来轮询项目仓库，使用 `getLatestRevision()` 方法来查看改动。

下面的例子程序监视了一个给定的项目仓库并且当某人提交了一个改动的时候打印日志消息。它一分钟只检查一次，所以不会对 Subversion 服务器造成太大的压力。

```
import org.tmatesoft.svn.core.*;
import org.tmatesoft.svn.core.io.*;
import org.tmatesoft.svn.core.internal.io.dav.*;
import java.util.*;
```

```
public class DetectChanges
{
  public static void main(String[] args)
          throws SVNException
  {
  String reposUrl = "http://svn.collab.net/repos/svn";
  SVNURL url = SVNURL.parseURIEncoded(reposUrl);

  DAVRepositoryFactory.setup();
  SVNRepository repository = SVNRepositoryFactory.create(url);

 long lastSeenRevision = repository.getLatestRevision();
 while(true) {
  long latestRevision = repository.getLatestRevision();
  if(latestRevision != lastSeenRevision) {
    displayChanges(lastSeenRevision + 1,
                   latestRevision, repository);
    lastSeenRevision = latestRevision;
   }
   pause(60);
  }
}
private static void displayChanges(long startRev, long endRev,
                                   SVNRepository repository)
        throws SVNException {
    String[] targetPaths = { "/" };
    List<SVNLogEntry> entries = new ArrayList<SVNLogEntry>();
    repository.log(targetPaths, entries, startRev,
                   endRev, false, false);
    for (SVNLogEntry entry : entries) {
      System.out.println("New revision " + entry.getRevision()
                         + " by " + entry.getAuthor());
      System.out.println("Log message: " + entry.getMessage());
   }
}
private static void pause(int seconds) {
   try {
     Thread.sleep(seconds * 1000);
   } catch (InterruptedException e) {
      // Do nothing
    }
  }
}
```

java/DetectChanges.java

如果不想只是打印日志消息的话，你可以把程序修改成发送邮件，运行构建脚本，或者任何其它你认为有用的东西。

■ 管理工作拷贝

JavaSVN 让你以与 svn 命令行工具相同的方式访问工作拷贝。特别是，SVNWCClient 类提供了 doXYZ()方法来映射命令行客户端的相同功能。比如说，我们可以使用 doInfo()方法来发出 svn info 命令：

```
import org.tmatesoft.svn.core.*;
import org.tmatesoft.svn.core.wc.*;
import java.io.File;
public class WorkingCopyInfo
{
  public static void main(String[] args)
          throws SVNException
  {
     File workingCopyRoot = new File("c:\\work\\subversion");
     SVNWCClient wcClient = new SVNWCClient(null, null);
     SVNInfo info = wcClient.doInfo(workingCopyRoot,
                                    SVNRevision.WORKING);
     System.out.println("Working Copy Info for " +
                        workingCopyRoot);
     System.out.println("URL: " +
                        info.getURL());
     System.out.println("Repository root: " +
                        info.getRepositoryRootURL());
     System.out.println("Last Changed Author: " +
                        info.getAuthor());
     System.out.println("Last Changed Rev: " +
                        info.getCommittedRevision());
     System.out.println("Last Changed Date: " +
                        info.getCommittedDate());
     System.out.println("URL: " +
                        info.getURL());
  }
}
```

java/WorkingCopyInfo.java

运行这个例子会产生类似于普通的 `svn info` 的输出：

```
Working Copy Info for c:\work\subversion
URL: http://svn.collab.net/repos/svn/trunk/subversion
Repository root: http://svn.collab.net/repos/svn
Last Changed Author: mbk
Last Changed Rev: 19040
Last Changed Date: Sun Mar 26 08:26:13 MST 2006
URL: http://svn.collab.net/repos/svn/trunk/subversion
```

`SVNWCClient` 允许你执行包括添加，删除，加锁，解锁在内的操作，但是你需要使用 `SVNCommitClient` 来把改动从工作拷贝提交到项目仓库。

希望这次对 JavaSVN 功能的浮光掠影的介绍能够给你一些印象，能大概知道一些关于如何把对 Subversion 的支持内嵌到你自己的程序中去的事情。如果你不是用 Java 写代码，记住还有很多语言可以供你选择。

D.2 高级项目仓库管理

在一个组织内设置 Subversion 时，一个常见的问题是“我应该创建多少个项目仓库？”我们的建议是只创建一个项目仓库，直到你有更多的具体需求出现时才创建更多的项目仓库。我们采用这种方式是因为在有需求的时候把现有的项目仓库分割成为两个不是难事。你拥有的项目仓库越多，管理工作就越繁重，备份啊，管理用户啊，诸如此类的琐事。因为 Subversion 对于分割，合并，辨认项目仓库有着良好的支持，所以当你创建了多个项目仓库之后，指不定哪天又要把它们合并到一起去。本节就是讲如何执行这些高级的项目仓库操作的。

■ 分割项目仓库

首先，确保你告诉了每个人你要分割项目仓库了。理想的情况是每个人都能签入，然后晚上下班回家，你一个人留下来把东西给组织好，第二天开始干一些新东西。如果人们不能提交他们的改动，你可能在你完成了分支之后，需要帮助他们重定向[1]他们的工作拷贝。一旦所有人提交了他们的改动，就可切断项目仓库的网络连接，以确保没有人再提交新的改动。这可能有些做得太过了，怎样处理取决于你的具体情况。但是偏安全一些不是坏事。

接下来，使用 `svnadmin dump` 来备份你的项目仓库，创建一个备份文件。第 176 页的 A.6 节“备份你的项目仓库”中讲过如何去做。确信你执行的是一个完整备份，而不是增量备份。一个备份文件是 Subversion 项目仓库的可移动的替身，我们可以用来在任何地方用它来重新创建项目仓库。

```
home> svnadmin dump /home/svnroot/log4rss > log4rss.dump
* Dumped revision 0.
* Dumped revision 1.
   :      :      :
* Dumped revision 37.
* Dumped revision 38.
```

[1] `svn switch` 命令包含一个`--relocate`选项，可以把旧的工作拷贝指向到新的服务器位置。

下一步要把备份文件载入到新的项目仓库之中，你需要创建并初始化它：

```
home> mkdir tools-repos
home> svnadmin create tools-repos
```

备份文件包含了你项目仓库中所有文件的完整历史。对于新的 tools 项目仓库，我们只对项目仓库中的某些路径感兴趣，比如说 log4rss/trunk/tools。使用 svndumpfilter 命令选择你想要移动到新项目仓库的目录，然后把它的输出管道连接到 svnadmin load 命令。

```
home> cat log4rss.dump                                    \
      | svndumpfilter include log4rss/trunk/tools \
      | svnadmin load tools-repos
Including prefixes:
   '/log4rss/trunk/tools'
Revision 0 committed as 0.
Revision 1 committed as 1.
Revision 2 committed as 2.
    :     :     :
<<< Started new transaction, based on original revision 38
------- Committed revision 38 >>>
```

svndumpfilter 会非常的唠叨，把 filter 包含进来的条目和抛弃掉的条目通通都给列出来。现在新的 tools-repos 项目仓库包含的只是 tools 目录了。

此时，你可以把新的项目仓库投入使用了，并且告诉开发人员去哪里找它。从原来的项目仓库中删掉 log4rss/trunk/tools 目录也是聪明的办法，这样的话人们就不会偶然地使用到过期的东西。Subversion 没有一个“毁尸灭迹”的命令，所以 tools 目录仍然会在旧的项目仓库中占用空间——如果这是一个问题，你可能需要考虑把你的备份文件载入到一个新的项目仓库中，使用“exclude”命令把你不再想要的目录给清除出去。在大部分情况下，这不会是一个问题，但是如果你的项目仓库包含大量的大文件的话，那就可能需要做一下这样的工作了。

■ 合并两个项目仓库

在某些情况下，你可能希望把两个现有的项目仓库合并到一起。一个可能的例子是两个独立的项目团队合并成了一个，或者一个新的项目团队取代

了一个现有团队的工作，并且想要使用他们自己的项目仓库来管理代码。

把一个项目仓库合并到另外一个中很简单，只要创建“donor”项目仓库的一个备份，使用 `svnadmin load` 把备份载入到目标项目仓库中去。载入的过程会回放在旧的项目仓库中发生过的每一个操作，所以虽然版本号不一样了，但是历史记录都还在。

这样的合并只在两个项目仓库有不同的目录结构时才行得通——如果目录是两个项目仓库都有的，载入过程就会失败。要避免发生这个问题，使用`--parent-dir` 选项来把代码载入到不同的位置。

例如，我们可以把 Log4RSS 项目仓库自己的备份载入到一个新的合并过的目录：

```
svnroot> svn mkdir file:///home/svnroot/log4rss/merge \
           -m "Create merge directory"
Committed revision 36.
svnroot> svnadmin load --parent-dir merge log4rss < log4rss.dump
<<< Started new transaction, based on original revision 1
    * adding path : merge/trunk ... done.
------- Committed new rev 37 (loaded from original rev 1) >>>
    :     :     :
<<< Started new transaction, based on original revision 25
    * editing path : merge/trunk/build.xml ... done.
------- Committed new rev 61 (loaded from original rev 25) >>>
```

在上面的示例中，你可以看到载入命令用原来的版本 25 创建了版本 61，而且没有编辑 `trunk/build.xml`，而是把它放到了 `merge/trunk/build.xml`，因此避免了和已经存在于项目仓库中的文件相冲突。

■ 组织项目仓库

在合并了两个项目仓库之后，你可能想要把东西重新组织一下，特别是你为了避免冲突使用了`--parent-dir` 选项之后。出于其他考虑，你可能也会想要重新整一整——可能项目仓库放在那很久了，一个都不是你想要的结构。幸运的是，Subversion 很擅长于把东西移来移去。

在 Subversion 中移动目录之前，确保你的所有用户已经把他们的改动都签入了。在把项目仓库乾坤大挪移之后，重新签出一个新的工作拷贝可能比在之前的版本上执行更新要容易一些。由于这个原因，你可能需要在周末做

这样的重新整理的工作，或者找一天大家都下班了的晚上。

使用项目仓库的 URL 作为 `svn mv` 的参数而不是使用工作拷贝，可以让移动立刻发生在服务器端。如果你的目录有大量的文件的话，非常有必要这么做。

我们推荐使用像 TortoiseSVN 这样的图形客户端来做大型项目仓库的整理工作，因为这样更容易跟踪目录的结构。Tortoise 的项目仓库浏览器让你可以通过拖拽文件和目录来移动他们——确实是非常方便！

PDG

附录 E

命令汇总和实例指导列表

Command Summary and Recipes

E.1 Subversion 命令汇总

大部分 Subversion 命令有共同的选项，我们首先把它们列出来，以避免在后面讲每个命令时不断重复。如果你不确定某个命令都支持哪些选项，只要运行 `svn help` *command* 就能查看到一个快速的摘要。

共同选项：

`--targets`	*list*	读取 *list* 并将其解释为一个将要操作的参数列表。
`--non-recurisive,-N`		只操作单个目录；不要处理子目录。
`--verbose, -v`		打印额外的信息。
`--quiet, -q`		打印的信息尽可能地少。
`--username`	*name*	指定在连接授权时使用的用户名。
`--password`	*pawd*	指定要使用的密码。
`--no-auth-cache`		不要缓存身份令牌。
`--non-interactive`		不要提示输入额外的信息。
`--config-dir`	*dir*	从 dir 读取用户配置。
`--editor-cmd`	*cmd*	使用 *cmd* 作为日志消息的编辑器。

svn add

把文件以及目录的名称添加给版本控制系统。他们会在下次提交时被添加到项目仓库之中去。

```
svn add path…
```

选项：

`--auto-props`	在添加它们的时候自动设置文件的属性。
`--no-auto-props`	禁用自动设置属性。

svn blame （也被称为 ann，annotate，praise）

显示文件每行的版本以及作者信息。

```
svn blame target…
```

选项：

`--revision, -r rev`	如果指定的 *rev* 是单个版本，显示该版本的作者信息。如果指定的是版本范围 *rev1:rev2*，显示 *rev2* 版本的作者信息，但是只检查版本到 *rev1*（要使得这命令有意义，*rev1* 必须小于 *rev2*）。

svn cat

输出指定文件或者 URL 的内容。

```
svn cat target…
```

选项：

`--revision, -r rev` 输出版本 *rev* 的 *target* 的内容。

svn checkout （也被称作 co）

从项目仓库签出一个工作拷贝。

```
svn checkout url… path
```

签出给定 URL。如果没有指定 path 参数，签出的本地目录名使用 URL 的 base name。如果指定了 path，而且 URL 参数只有一个，签出到该目录。如果指定了 path，但是有多个 URL 参数，签出到 path 目录之下的子目录中，目录名使用对应 URL 的 base name。

选项：

`--revision, -r rev` 要签出的版本。

svn cleanup

清理工作拷贝，移除锁，完成未完成的操作，等等。

```
svn cleanup path…
```

svn commit （也被称作 ci）

把改动从你的工作拷贝发送到项目仓库。

```
svn commit path…
```

选项：

`--message,`	`-m msg`	使用 *msg* 作为提交日志消息。
`--file,`	`-F file`	使用 *file* 的内容做为提交日志消息。
`--no-unlock`		不要在提交的时候释放锁。

svn copy （也被称作 cp）

在工作拷贝或者项目仓库中制造包括历史在内的复本。

```
svn copy src dest
```

src 和 *dest* 可以是工作拷贝（WC）的路径或者 URL。

src	dest	效果……
WC	WC	拷贝并添加（包括历史）。
WC	URL	立即提交 WC 的拷贝到 URL。
URL	WC	签出 URL 到 WC，添加。
URL	URL	完全服务器端拷贝；用于制作分支和打标签。

选项：

`--revision, -r rev` 要拷贝的 *src* 的版本。只在 *src* 是项目仓库的 URL 时才有意义。

svn delete （也被称作 del, remove, rm）

从版本控制系统中移除文件或者目录。

```
svn delete target…
```

从项目仓库删除文件或者目录。如果 *target* 是工作拷贝中的文件或者目录，它被从工作拷贝中移除并且预计在下次提交时删除掉。如果 *target* 是项目仓库 URL，通过一次立即的提交从项目仓库中删除。

选项：

`--message,`	`-m msg`	使用 *msg* 作为提交日志消息。
`--file,`	`-F file`	使用 *file* 的内容作为提交日志消息。

svn diff （也被称作 di）

显示两个路径之间的差异。

```
svn diff -r rev1:rev2 target…
svn diff oldurl newurl
```

在第二种格式中，显示 *target* 在两个版本 *rev1* 和 *rev2* 之间的改动。target 可以是工作拷贝路径或者 URL。

在第二种格式中，显示最新版本的 *oldurl* 和 *newurl* 之间的差异。

选项：

`--old arg` 使用 *arg* 作为老一些的目标。

`--new arg` 使用 *arg* 作为新一些的目标。

svn export

创建树的一个无版本记录的拷贝。

```
svn export -rrev URL path
```

从项目仓库的指定 URL 导出一个干净的目录树到 *path* 中，如果指定了 *rev* 参数，导出 *rev* 版本的，否则导出最新版本。如果 *path* 被省略了，URL 的最后一节被用作本地目录的名称。

选项：

`--revision, -r rev` 导出项目仓库的 *rev* 版本。

`--native-eol style` 对于有本地 svn:eol-style 属性的文件，使用不同的行结尾标识符而不是系统标准的文件行结尾标识符。*style* 必须是 LF，CR 或者 CRLF 之中的一个。

svn import

提交一个无版本的文件或者树到项目仓库。

```
svn import path URL
```

递归提交 *path* 的一个拷贝到 *URL*。如果 *path* 被省略了，就假定要提交的目录为当前目录。父目录会根据需要在项目仓库中创建。

选项：

`--auto-props` 在导入过程中自动设置属性给文件。

`--no-auto-props` 对于导入的文件禁用自动设置属性。

svn info

显示文件或者目录的信息。

```
svn info path…
```

打印每个 *path* 的信息。

选项：

`--recursive, -r` 递归下降

svn list （也被称作 `ls`）

列出项目仓库中的目录条目。

```
svn list target…
```

如果是在项目仓库中，列出每个 *target* 文件以及每个 *target* 目录的内容。如果 *target* 是在工作拷贝中，会使用对应的项目仓库 URL。

选项：

`--verbose, -v` 显示每个目录条目的额外信息。

svn lock

锁住文件让其他用户不能提交改动。

```
svn lock target
```

与项目仓库服务器联系，获得对一个或者多个工作拷贝文件的锁。一旦锁住了，其他用户就不能提交改动给文件，除非锁被释放或者被砸坏。

选项：

`--message, -m msg` 使用 *msg* 作为锁信息消息。

`--force` 强制加锁成功，通过从其他用户或者工作拷贝把锁给偷过来。

svn log

显示一些版本或者文件的日志消息。

```
svn log target
```

打印本地路径或者项目仓库 URL 的日志消息。对于每个本地路径，默认的版本范围是 BASE:1，对于 URL 默认的版本范围是 HEAD:1。

选项：

`--revision, -r rev`	如果 *rev* 是单个版本，只显示该版本的日志条目。 如果 *rev* 是版本范围，显示这些版本的日志条目。
`--verbose, -v`	打印每条日志消息所影响的路径。
`--stop-on-copy`	在遍历历史的时候不要穿越拷贝（对于查找分支的起点很有用）。

svn merge

把两个来源的差异应用给工作拷贝路径。

```
svn merge sourceURL1@rev1 sourceURL2@rev2 wcpath
svn merge sourceWCPATH1@rev1 sourceWCPATH2@rev2 wcpath
svn merge -r rev1:rev2 SOURCE wcpath
```

第一个格式中，源 URL 被指定了版本 *rev1* 和 *rev2*.这是两个要被比较的源。如果没有指明版本，默认使用最新版本。

在第二种格式中，对应工作拷贝的 URL 被指定做为要比较源。版本号必须指定。

在第三种格式中，SOURCE 可以是 URL 或者工作拷贝中条目，使用工作拷贝的话会使用对应的 URL 替代。这个 URL 会被比较它的 *rev1* 和 *rev2* 版本。

wcpath 是接收这些改动的工作拷贝路径。如果 *wcpath* 被省略，假定就是当前目录，除非来源有一个唯一的 basename 匹配了当前目录的文件，如果是这样的话改动就会应用于那个文件。

选项：

`--diff3-cmd cmd`	使用 *cmd* 作为合并命令。
`--ignore-ancestry`	在计算合并时忽略过去历史。

svn mkdir

创建版本控制下的新目录。

```
svn mkdir target…
```

用工作拷贝路径指定的每个目录会在本地被创建并且预计在下次提交时被添加。用 URL 指定的每个路径通过一次立即提交直接在项目仓库创建好。

两种情况中，所有的间接目录都必须存在。

svn move（也被称作 `mv`，`rename`，`ren`）

移动或者重命名工作拷贝或者项目仓库中的文件或者目录。

```
svn move src dest
```

src 和 *dest* 必须是工作拷贝路径或者项目仓库 URL。在工作拷贝中，移动被执行并且新的位址被计划于下次提交时添加。对于项目仓库 URL，一个完全服务器端的重命名操作被立即执行。

选项：

`--revision, -r rev` 使用版本 *rev* 作为源来执行这次移动。

svn propdel（也被称作 `pdel`，`pd`）

删除文件或者目录的属性。

```
svn propdel propname path…
```

在本地工作拷贝中删除 *path* 的 *propname* 属性。

svn propedit（也被称作 `pedit`，`pe`）

编辑文件或者目录的属性。

```
svn propedit propname path…
```

打开一个外部的编辑器在本地工作拷贝中编辑 *path* 的 *propname* 属性。

svn propget（也被称作 `pget`，`pg`）

打印文件或者目录的属性值。

```
svn propget propname path…
```

打印每个 *path* 的 *propname* 的内容。默认情况下，Subversion 会在属性值的结尾添加一个额外的换行，这样输出看起来漂亮些。而且，无论何时包括了多个路径，每个属性值都会用关联路径作为前缀标识出来。

选项：

`--strict` 禁用额外的换行和其他的美化措施（在把二进制属性重定向到文件时会有用处）。

svn proplist（也被称作 plist，pl）

列出文件或者目录的所有属性。

svn proplist *path…*

列出 path 的所有属性。

选项：

--verbose,	-v	打印额外的信息。
--recursive,	-R	递归下降。
--revision,	-r *rev*	列出 *path* 在版本 *rev* 定义的属性。

svn propset（也被称作 pset，ps）

svn propset propname propval path…

对于 *path*，设置值 *propval* 给 *propname* 属性。如果 *propval* 没有指定，你必须使用-F 选项来指定一个文件，其内容会作为属性的值设置给属性。

选项：

--file,	-F *file*	读取 *file* 的内容，使用它作为属性值。
--recursive,	-R	递归下降
--encoding	*enc*	把值作为用 *enc* 编码的字符集。

svn resolved

移除工作拷贝文件或者目录的冲突状态。

svn resolved *path…*

标记之前包含冲突的文件为“已解决”。注意这个命令不会从语意上解决冲突或者移除冲突标记；它只是移除冲突相关的文件并且允许 *path* 被再次提交。

选项：

--recursive, -R 递归下降。

svn revert

恢复工作拷贝中的文件（撤销最新的本地修改）。

```
svn revert path…
```

这个命令不需要网络连接并且撤销你对 *path* 所做的一切改动。它不恢复被删除的目录。

选项：

`--recursive, -R` 递归下降。

svn status（也被称作 `stat, st`）

打印工作拷贝中文件和目录的状态

```
svn status path…
```

没有任何参数，只是打印本地修改过的条目（无须访问网络）。

加-u 参数，添加工作版本和服务器过期信息。

加-v 参数，打印每个条目完整的版本信息。

输出中的前六列的宽度均为一个字符。

第一列：指出条目是被添加了、删除还是修改过了。

“ ”	无改动。
A	添加了。
C	冲突了。
D	删除了。
G	合并了。
I	忽略了。
M	修改了。
R	替换了。
X	条目无版本记录，但是被外部定义使用了。
?	条目未用版本控制管理。
!	找不到条目（没有用 svn 命令正常移除）或者不完整。
~	有版本的条目被某种其他条目给阻碍了。

第二列：文件或者目录属性的修改状态。

“ ”	无改动。
C	冲突了。
M	修改了。

第三列：工作拷贝目录是否被锁住了。

“ ”	未锁住。
L	锁住了。

第四列： 计划中的提交会包含带有历史的添加操作。

“ ” 没有历史预计被提交。

+ 历史预计被提交。

第五列： 条目是否被转向为相对于其上级条目。

“ ” 正常。

S 转向了。

第六列： 锁令牌信息（要显示项目仓库信息，使用-u）。

“ ” 没有锁令牌，项目仓库未被加锁。

K 锁令牌存在，条目在项目仓库中被锁住了。

O 条目在项目仓库中被锁住了，但是锁令牌在其他工作拷贝中。

T 条目在项目仓库中被锁住了，锁令牌在工作拷贝中，但是被其他工作拷贝给偷走了。

B 条目没有在项目仓库中锁住，但是被砸坏了的锁令牌在工作拷贝中。

过期信息出现在第八列（用-u 选项获得）。

* 更新的版本存在于服务器中。

“ ” 工作拷贝是最新的。

剩下域的长度是不断变化的，由空格分开：工作拷贝的版本（用-u 或者-v），最后提交的版本，最后提交的作者（用-v）。工作拷贝的路径总是最后一个域，所以它可以包含空格。

选项：

--show-updates, -u 联系服务器显示更新信息。

--verbose, -v 打印额外的信息。

--non-recursive, -N 只操作单个目录。

--no-ignore 忽视默认设置和 svn:ignore 属性设置的忽略项。

svn switch（也被称作 sw）

把工作拷贝转向到其他 URL。

```
svn switch URL path
```

更新工作拷贝让其使用项目仓库的新 URL。这个行为类似 svn update 而且是一种把工作拷贝转向到同一项目仓库中的分支或者标签的办法。

选项：

```
--revision,      -r  rev  转向到版本 rev。
--non-recursive, -N       只操作单个目录。
--diff3-cmd           cmd 使用 cmd 做为合并命令。
```

svn unlock

解开工作拷贝文件或者项目仓库 URL 的锁。

```
svn unlock target…
```

释放当前对 *target* 的锁，以让其他用户可以提交改动。

选项：

```
--force 砸坏现有对 target 的锁,甚至它不是被当前工作拷贝所拥有的。
```

svn update（也被称作 up）

把改动从项目仓库带到工作拷贝来。

```
svn update path…
```

如果没有指定版本，把工作拷贝更新为最新版本。否则把工作拷贝同步为用-r 选项指定的版本。

对于每个更新过的条目会有单独一行，开头有一个字符表示做过的动作。这些字符有以下含义：

A　添加了。

D　删除了。

U　更新了。

C　冲突了。

M　合并了。

第一列的字符表示了对实际文件的更新，对于文件属性的更新被列在第二列。

选项：

`--revision, -r`	*rev*	更新到版本 *rev*。
`--non-recrusive, -N`		只操作单个目录。
`--diff3-cmd`	*cmd*	使用 *cmd* 作为合并命令。

E.2 实例指导列表

附录 F

其他资源

Other Resources

与 Subversion 和版本控制相关的资源有很多。这儿列出了其中一小部分基本的。

F.1 在线资源

Subversion 主页…http://subversion.tigris.org/

官方的 Subversion 网站对于所有刚开始学 Subversion 的人是极好的资源。该网站有各种文档，其中有很棒的 Subversion FAQ，它回答了大部分常见的问题。项目链接页面是寻找 Subversion 相关软件，插件，文章和文档的好地方。

你还可以加入 Subversion 的用户邮件列表：只要发送邮件到 users-subscribe@subversion.tigris.org。这个列表是问问题的地方，经常可以遇到一些很友好的人，Subversion 的核心开发者也在这个列表中。

Pragmatic Programmers…http://www.pragmaticprogrammer.com/titles/svn/

本书的配套网站，你可以在其中找到样例代码，勘误，以及其他有用的链接。

Subversion Book…http://svnbook.red-bean.com/

官方的 Subversion book 可以在线阅读或者打印，包含了很深入的讨论，甚至对于 Subversion 最隐秘的特性也有涉及。

Better SCM…http://better-scm.berlios.de/

Better SCM 项目致力于推动 CVS 的替代品的应用，包含了数个版本控制系统的比较。

CM Crossroads…http://www.cmcrossroads.com/

配置管理是一个比版本控制更大的话题，但是要达到其目标，一般都需要良好的版本控制作为支撑。本书中很多地方都提到了“SCM 模式”的名字，所以如果你想要更深入地了解这些模式或者对于如何组织源代码、构建、项目和发布感兴趣，这个网站中有你想要的文章和讨论组。

F.2 参考书目

[BA03] Stephen P. Berczuk and Brad Appleton. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration.* Addison-Wesley, 2003.

[Cla04] Mike Clark. *Pragmatic Project Automation. How to Build, Deploy, and Monitor Java Applications*（《项目自动化之道——如何构建、部署、监控Java应用》）. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2004.

[HT03] Andrew Hunt and David Thomas. *Pragmatic Unit Testing In Java with Junit*（《单元测试之道Java版——使用JUnit》）. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2003.

[HT04] Andrew Hunt and David Thomas. *Pragmatic Unit Testing In C# with Nunit*（《单元测试之道C#版——使用NUnit》）. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2004.

索引

Index

Symbols

A

B

C

G

H

I

J

K

L

T

U

V

W